HTML5 Canvas

Notes for Professionals



100+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Contents



About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/HTML5CanvasBook

This *HTML5 Canvas Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official HTML5 Canvas group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with HTML5 Canvas

Section 1.1: Detecting mouse position on the canvas

This example will show how to get the mouse position relative to the canvas, such that `(0,0)` will be the top-left hand corner of the HTML5 Canvas. The `e.clientX` and `e.clientY` will get the mouse positions relative to the top of the document, to change this to be based on the top of the canvas we subtract the `left` and `right` positions of the canvas from the client X and Y.

```
var canvas = document.getElementById("myCanvas");
var ctx = canvas.getContext("2d");
ctx.font = "16px Arial";

canvas.addEventListener("mousemove", function(e) {
    var cRect = canvas.getBoundingClientRect();        // Gets CSS pos, and width/height
    var canvasX = Math.round(e.clientX - cRect.left);  // Subtract the 'left' of the canvas
    var canvasY = Math.round(e.clientY - cRect.top);   // from the X/Y positions to make
    ctx.clearRect(0, 0, canvas.width, canvas.height);  // (0,0) the top left of the canvas
    ctx.fillText("X: "+canvasX+", Y: "+canvasY, 10, 20);
});
```

Runnable Example

The use of `Math.round` is due to ensure the `x,y` positions are integers, as the bounding rectangle of the canvas may not have integer positions.

Section 1.2: Canvas size and resolution

The size of a canvas is the area it occupies on the page and is defined by the CSS width and height properties.

```
canvas {
   width : 1000px;
   height : 1000px;
}
```

The canvas resolution defines the number of pixels it contains. The resolution is set by setting the canvas element width and height properties. If not specified the canvas defaults to 300 by 150 pixels.

The following canvas will use the above CSS size but as the `width` and `height` is not specified the resolution will be 300 by 150.

```
<canvas id="my-canvas"></canvas>
```

This will result in each pixel being stretched unevenly. The pixel aspect is 1:2. When the canvas is stretched the browser will use bilinear filtering. This has an effect of blurring out pixels that are stretched.

For the best results when using the canvas ensure that the canvas resolution matches the display size.

Following on from the CSS style above to match the display size add the canvas with the `width` and `height` set to the same pixel count as the style defines.

```
<canvas id = "my-canvas" width = "1000" height = "1000"></canvas>
```

Section 1.3: Rotate

The `rotate(r)` method of the 2D context rotates the canvas by the specified amount `r` of *radians* around the origin.

HTML

```
<canvas id="canvas" width=240 height=240 style="background-color:#808080;">
</canvas>

<button type="button" onclick="rotate_ctx();">Rotate context</button>
```

JavaScript

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
var ox = canvas.width / 2;
var oy = canvas.height / 2;
ctx.font = "42px serif";
ctx.textAlign = "center";
ctx.textBaseline = "middle";
ctx.fillStyle = "#FFF";
ctx.fillText("Hello World", ox, oy);

rotate_ctx = function() {
  // translate so that the origin is now (ox, oy) the center of the canvas
  ctx.translate(ox, oy);
  // convert degrees to radians with radians = (Math.PI/180)*degrees.
  ctx.rotate((Math.PI / 180) * 15);
  ctx.fillText("Hello World", 0, 0);
  // translate back
  ctx.translate(-ox, -oy);
};
```

Live demo on JSfiddle

Section 1.4: Save canvas to image file

You can save a canvas to an image file by using the method `canvas.toDataURL()`, that returns the *data URI* for the canvas' image data.

The method can take two optional parameters `canvas.toDataURL(type, encoderOptions)`: `type` is the image format (if omitted the default is `image/png`); `encoderOptions` is a number between 0 and 1 indicating image quality (default is 0.92).

Here we draw a canvas and attach the canvas' data URI to the "Download to myImage.jpg" link.

HTML

```
<canvas id="canvas" width=240 height=240 style="background-color:#808080;">
</canvas>
<p></p>
<a id="download" download="myImage.jpg" href="" onclick="download_img(this);">Download to
myImage.jpg</a>
```

JavaScript

```
var canvas = document.getElementById("canvas");
```

```
var ctx = canvas.getContext("2d");
var ox = canvas.width / 2;
var oy = canvas.height / 2;
ctx.font = "42px serif";
ctx.textAlign = "center";
ctx.textBaseline = "middle";
ctx.fillStyle = "#800";
ctx.fillRect(ox / 2, oy / 2, ox, oy);

download_img = function(el) {
  // get image URI from canvas object
  var imageURI = canvas.toDataURL("image/jpg");
  el.href = imageURI;
};
```

Live demo on JSfiddle.

Section 1.5: How to add the Html5 Canvas Element to a webpage

Html5-Canvas ...

- Is an Html5 element.
- Is supported in most modern browsers (Internet Explorer 9+).
- Is a visible element that is transparent by default
- Has a default width of 300px and a default height of 150px.
- Requires JavaScript because all content must be programmatically added to the Canvas.

Example: Create an Html5-Canvas element using both Html5 markup and JavaScript:

```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvasHtml5{border:1px solid red; }
    #canvasJavascript{border:1px solid blue; }
</style>
<script>
window.onload=(function(){

    // add a canvas element using javascript
    var canvas=document.createElement('canvas');
    canvas.id='canvasJavascript'
    document.body.appendChild(canvas);

}); // end $(function(){});
</script>
</head>
<body>

    <!-- add a canvas element using html -->
    <canvas id='canvasHtml5'></canvas>

</body>
</html>
```

Section 1.6: An index to Html5 Canvas Capabilities & Uses

Capabilities of the Canvas

Canvas lets you programmatically draw onto your webpage:

- Images,
- Texts,
- Lines and Curves.

Canvas drawings can be extensively styled:

- stroke width,
- stroke color,
- shape fill color,
- opacity,
- shadowing,
- linear gradients and radial gradients,
- font face,
- font size,
- text alignment,
- text may be stroked, filled or both stroked & filled,
- image resizing,
- image cropping,
- compositing

Uses of the Canvas

Drawings can be combined and positioned anywhere on the canvas so it can be used to create:

- Paint / Sketch applications,
- Fast paced interactive games,
- Data analyses like charts, graphs,
- Photoshop-like imaging,
- Flash-like advertising and Flashy web content.

Canvas allows you to manipulate the Red, Green, Blue & Alpha component colors of images. This allows canvas to manipulate images with results similar to Photoshop.

- Recolor any part of an image at the pixel level (if you use HSL you can even recolor an image while retaining the important Lighting & Saturation so the result doesn't look like someone slapped paint on the image),
- "Knockout" the background around a person/item in an image,
- Detect and Floodfill part of an image (eg, change the color of a user-clicked flower petal from green to yellow -- just that clicked petal!),
- Do Perspective warping (e.g. wrap an image around the curve of a cup),
- Examine an image for content (eg. facial recognition),
- Answer questions about an image: Is there a car parked in this image of my parking spot?,
- Apply standard image filters (grayscale, sepia, etc)
- Apply any exotic image filter you can dream up (Sobel Edge Detection),
- Combine images. If dear Grandma Sue couldn't make it to the family reunion, just "photoshop" her into the reunion image. Don't like Cousin Phil -- just "photoshop him out,
- Play a video / Grab a frame from a video,
- Export the canvas content as a .jpg | .png image (you can even optionally crop or annotate the image and

export the result as a new image),

About moving and editing canvas drawings (for example to create an action game):

- After something has been drawn on the canvas, that existing drawing cannot be moved or edited. This common misconception that canvas drawings are movable is worth clarifying: *Existing canvas drawings cannot be edited or moved!*
- Canvas draws very, very quickly. Canvas can draw hundreds of images, texts, lines & curves in a fraction of a second. It uses the GPU when available to speed up drawing.
- Canvas creates the illusion of motion by quickly and repeatedly drawing something and then redrawing it in a new position. Like television, this constant redrawing gives the eye the illusion of motion.

Section 1.7: Off screen canvas

Many times when working with the canvas you will need to have a canvas to hold some intrum pixel data. It is easy to create an offscreen canvas, obtain a 2D context. An offscreen canvas will also use the available graphics hardware to render.

The following code simply creates a canvas and fills it with blue pixels.

```
function createCanvas(width, height){
    var canvas = document.createElement("canvas"); // create a canvas element
    canvas.width = width;
    canvas.height = height;
    return canvas;
}

var myCanvas = createCanvas(256,256); // create a small canvas 256 by 256 pixels
var ctx = myCanvas.getContext("2d");
ctx.fillStyle = "blue";
ctx.fillRect(0,0,256,256);
```

Many times the offscreen canvas will be used for many tasks, and you may have many canvases. To simplify the use of the canvas you can attach the canvas context to the canvas.

```
function createCanvasCTX(width, height){
    var canvas = document.createElement("canvas"); // create a canvas element
    canvas.width = width;
    canvas.height = height;
    canvas.ctx = canvas.getContext("2d");
    return canvas;
}
var myCanvas = createCanvasCTX(256,256); // create a small canvas 256 by 256 pixels
myCanvas.ctx.fillStyle = "blue";
myCanvas.ctx.fillRect(0,0,256,256);
```

Section 1.8: Hello World

HTML

```
<canvas id="canvas" width=300 height=100 style="background-color:#808080;">
</canvas>
```

JavaScript

```
var canvas = document.getElementById("canvas");
```

```
var ctx = canvas.getContext("2d");
ctx.font = "34px serif";
ctx.textAlign = "center";
ctx.textBaseline="middle";
ctx.fillStyle = "#FFF";
ctx.fillText("Hello World",150,50);
```

Result

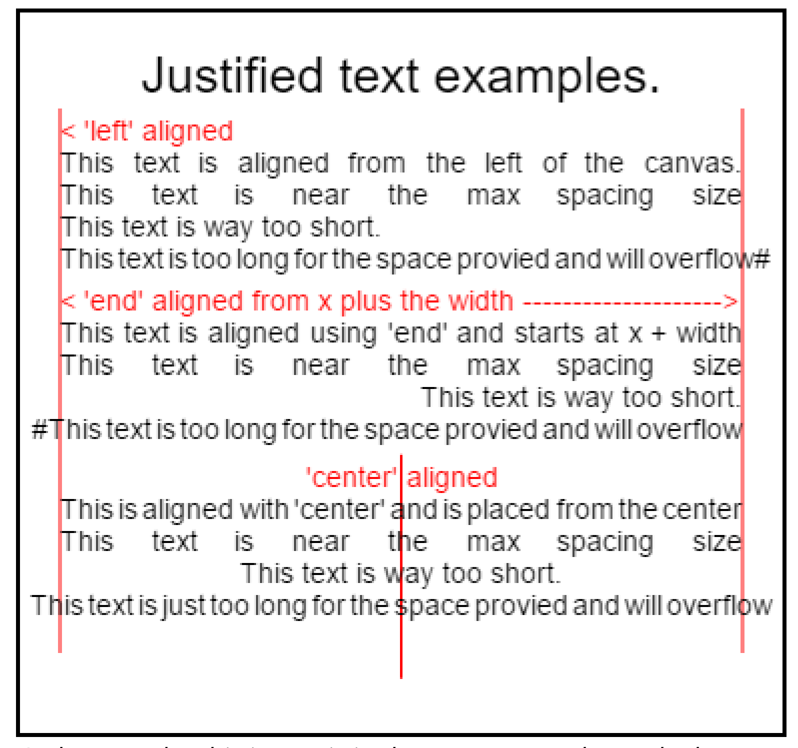
Hello World

Chapter 2: Text

Section 2.1: Justified text

This example renders justified text. It adds extra functionality to the `CanvasRenderingContext2D` by extending its prototype or as a global object `justifiedText` (optional see Note A).

Example rendering.



Code to render this image is in the usage examples at the bottom.

The Example

The function as a anonymous immediately invoked function.

```
(function(){
    const FILL = 0;        // const to indicate filltext render
    const STROKE = 1;
    const MEASURE = 2;
    var renderType = FILL; // used internal to set fill or stroke text

    var maxSpaceSize = 3; // Multiplier for max space size. If greater then no justificatoin applied
    var minSpaceSize = 0.5; // Multiplier for minimum space size
    var renderTextJustified = function(ctx,text,x,y,width){
        var words, wordsWidth, count, spaces, spaceWidth, adjSpace, renderer, i, textAlign,
useSize, totalWidth;
        textAlign = ctx.textAlign; // get current align settings
        ctx.textAlign = "left";
        wordsWidth = 0;
        words = text.split(" ").map(word => {
            var w = ctx.measureText(word).width;
            wordsWidth += w;
            return {
                width : w,
                word : word,
```

```
            };
        });
        // count = num words, spaces = number spaces, spaceWidth normal space size
        // adjSpace new space size >= min size. useSize Resulting space size used to render
        count = words.length;
        spaces = count - 1;
        spaceWidth = ctx.measureText(" ").width;
        adjSpace = Math.max(spaceWidth * minSpaceSize, (width - wordsWidth) / spaces);
        useSize = adjSpace > spaceWidth * maxSpaceSize ? spaceWidth : adjSpace;
        totalWidth = wordsWidth + useSize * spaces
        if(renderType === MEASURE){ // if measuring return size
            ctx.textAlign = textAlign;
            return totalWidth;
        }
        renderer = renderType === FILL ? ctx.fillText.bind(ctx) : ctx.strokeText.bind(ctx); // fill
or stroke
        switch(textAlign){
            case "right":
                x -= totalWidth;
                break;
            case "end":
                x += width - totalWidth;
                break;
            case "center": // intentional fall through to default
                x -= totalWidth / 2;
            default:
        }
        if(useSize === spaceWidth){ // if space size unchanged
            renderer(text,x,y);
        } else {
            for(i = 0; i < count; i += 1){
                renderer(words[i].word,x,y);
                x += words[i].width;
                x += useSize;
            }
        }
        ctx.textAlign = textAlign;
    }
    // Parse vet and set settings object.
    var justifiedTextSettings = function(settings){
        var min,max;
        var vetNumber = (num, defaultNum) => {
            num = num !== null && num !== null && !isNaN(num) ? num : defaultNum;
            if(num < 0){
                num = defaultNum;
            }
            return num;
        }
        if(settings === undefined || settings === null){
            return;
        }
        max = vetNumber(settings.maxSpaceSize, maxSpaceSize);
        min = vetNumber(settings.minSpaceSize, minSpaceSize);
        if(min > max){
            return;
        }
        minSpaceSize = min;
        maxSpaceSize = max;
    }
    // define fill text
    var fillJustifyText = function(text, x, y, width, settings){
        justifiedTextSettings(settings);
```

```
        renderType = FILL;
        renderTextJustified(this, text, x, y, width);
    }
    // define stroke text
    var strokeJustifyText = function(text, x, y, width, settings){
        justifiedTextSettings(settings);
        renderType = STROKE;
        renderTextJustified(this, text, x, y, width);
    }
    // define measure text
    var measureJustifiedText = function(text, width, settings){
        justifiedTextSettings(settings);
        renderType = MEASURE;
        return renderTextJustified(this, text, 0, 0, width);
    }
    // code point A
    // set the prototypes
    CanvasRenderingContext2D.prototype.fillJustifyText = fillJustifyText;
    CanvasRenderingContext2D.prototype.strokeJustifyText = strokeJustifyText;
    CanvasRenderingContext2D.prototype.measureJustifiedText = measureJustifiedText;
    // code point B

    // optional code if you do not wish to extend the CanvasRenderingContext2D prototype
    /* Uncomment from here to the closing comment
    window.justifiedText = {
        fill : function(ctx, text, x, y, width, settings){
            justifiedTextSettings(settings);
            renderType = FILL;
            renderTextJustified(ctx, text, x, y, width);
        },
        stroke : function(ctx, text, x, y, width, settings){
            justifiedTextSettings(settings);
            renderType = STROKE;
            renderTextJustified(ctx, text, x, y, width);
        },
        measure : function(ctx, text, width, settings){
            justifiedTextSettings(settings);
            renderType = MEASURE;
            return renderTextJustified(ctx, text, 0, 0, width);
        }
    }
    to here*/
})();
```

> **Note A:** If you do not wish to extend the `CanvasRenderingContext2D` prototype Remove from the example all code between `// code point A` and `// code point B` and uncomment the code marked `/* Uncomment from here to the closing comment`

How to use

Three functions are added to the `CanvasRenderingContext2D` and are available to all 2D context objects created.

- ctx.fillJustifyText(text, x, y, width, [settings]);
- ctx.strokeJustifyText(text, x, y, width, [settings]);
- ctx.measureJustifiedText(text, width, [settings]);

Fill and stroke text function fill or stroke text and use the same arguments. `measureJustifiedText` will return the

actual width that text would be rendered at. This may be equal, less, or greater than the argument `width` depending on current settings.

> **Note:** Arguments inside [and] are optional.

Function arguments

- **text:** String containing the text to be rendered.
- **x, y:** Coordinates to render the text at.
- **width:** Width of the justified text. Text will increase/decrease spaces between words to fit the width. If the space between words is greater than `maxSpaceSize` (default = 6) times normal spacing will be used and the text will not fill the required width. If the spacing is less than `minSpaceSize` (default = 0.5) time normal spacing then the min space size is used and the text will overrun the width requested
- **settings:** Optional. Object containing min and max space sizes.

The `settings` argument is optional and if not included text rendering will use the last setting defined or the default (shown below).

Both min and max are the min and max sizes for the [space] character separating words. The default `maxSpaceSize = 6` means that when the space between characters is > 63 * ctx.measureText(" ").width text will not be justified. If text to be justified has spaces less than `minSpaceSize = 0.5` (default value 0.5) * `ctx.measureText(" ").width` the spacing will be set to `minSpaceSize * ctx.measureText(" ").width` and the resulting text will overrun the justifying width.

The following rules are applied, min and max must be numbers. If not then the associate values will not be changed. If `minSpaceSize` is larger than `maxSpaceSize` both input setting are invalid and min max will not be changed.

Example setting object with defaults

```
settings = {
    maxSpaceSize : 6;   // Multiplier for max space size.
    minSpaceSize : 0.5; // Multiplier for minimum space size
};
```

> **NOTE:** These text functions introduce a subtle behaviour change for the `textAlign` property of the 2D context. 'Left', 'right', 'center' and 'start' behave as is expected but 'end' will not align from the right of the function argument `x` but rather from the right of `x + width`

> **Note:** settings (min and max space size) are global to all 2D context objects.

USAGE Examples

```
var i = 0;
text[i++] = "This text is aligned from the left of the canvas.";
text[i++] = "This text is near the max spacing size";
text[i++] = "This text is way too short.";
text[i++] = "This text is too long for the space provied and will overflow#";
```

```
text[i++] = "This text is aligned using 'end' and starts at x + width";
text[i++] = "This text is near the max spacing size";
text[i++] = "This text is way too short.";
text[i++] = "#This text is too long for the space provied and will overflow";
text[i++] = "This is aligned with 'center' and is placed from the center";
text[i++] = "This text is near the max spacing size";
text[i++] = "This text is way too short.";
text[i++] = "This text is just too long for the space provied and will overflow";

// ctx is the 2d context
// canvas is the canvas

ctx.clearRect(0,0,w,h);
ctx.font = "25px arial";
ctx.textAlign = "center"
var left = 20;
var center = canvas.width / 2;
var width = canvas.width-left*2;
var y = 40;
var size = 16;
var i = 0;
ctx.fillText("Justified text examples.",center,y);
y+= 40;
ctx.font = "14px arial";
ctx.textAlign = "left"
var ww = ctx.measureJustifiedText(text[0], width);
var setting = {
    maxSpaceSize : 6,
    minSpaceSize : 0.5
}
ctx.strokeStyle = "red"
ctx.beginPath();
ctx.moveTo(left,y - size * 2);
ctx.lineTo(left, y + size * 15);
ctx.moveTo(canvas.width - left,y - size * 2);
ctx.lineTo(canvas.width - left, y + size * 15);
ctx.stroke();
ctx.textAlign = "left";
ctx.fillStyle = "red";
ctx.fillText("< 'left' aligned",left,y - size)
ctx.fillStyle = "black";
ctx.fillJustifyText(text[i++], left, y, width, setting);  // settings is remembered
ctx.fillJustifyText(text[i++], left, y+=size, width);
ctx.fillJustifyText(text[i++], left, y+=size, width);
ctx.fillJustifyText(text[i++], left, y+=size, width);
y += 2.3*size;
ctx.fillStyle = "red";
ctx.fillText("< 'end' aligned from x plus the width -------------------->",left,y - size)
ctx.fillStyle = "black";
ctx.textAlign = "end";
ctx.fillJustifyText(text[i++], left, y, width);
ctx.fillJustifyText(text[i++], left, y+=size, width);
ctx.fillJustifyText(text[i++], left, y+=size, width);
ctx.fillJustifyText(text[i++], left, y+=size, width);

y += 40;
ctx.strokeStyle = "red"
ctx.beginPath();
ctx.moveTo(center,y - size * 2);
ctx.lineTo(center, y + size * 5);
ctx.stroke();
ctx.textAlign = "center";
```

```
ctx.fillStyle = "red";
ctx.fillText("'center' aligned",center,y - size)
ctx.fillStyle = "black";
ctx.fillJustifyText(text[i++], center, y, width);
ctx.fillJustifyText(text[i++], center, y+=size, width);
ctx.fillJustifyText(text[i++], center, y+=size, width);
ctx.fillJustifyText(text[i++], center, y+=size, width);
```

Section 2.2: Justified paragraphs

Renders text as justified paragraphs. **REQUIRES** the example **Justified text**

Example render

Justified paragraph examples.

Typesetting is the composition of text by means of arranging physical types or the digital equivalents. Stored letters and other symbols are retrieved and ordered according to a language's orthography for visual display. Typesetting requires the prior process of designing a font. One significant effect of typesetting was that authorship of works could be spotted more easily; making it difficult for copiers who have not gained permission.

Typesetting is the composition of text by means of arranging physical types or the digital equivalents. Stored letters and other symbols are retrieved and ordered according to a language's orthography for visual display. Typesetting requires the prior process of designing a font. One significant effect of typesetting was that authorship of works could be spotted more easily; making it difficult for copiers who have not gained permission.

Top paragraph has **setting.compact = true** and bottom **false** and line spacing is **1.2** rather than the default **1.5**. Rendered by code usage example bottom of this example.

Example code

```
 // Requires justified text extensions
(function(){
    // code point A
    if(typeof CanvasRenderingContext2D.prototype.fillJustifyText !== "function"){
        throw new ReferenceError("Justified Paragraph extension missing requiered
CanvasRenderingContext2D justified text extension");
    }
    var maxSpaceSize = 3; // Multiplier for max space size. If greater then no justificatoin applied
    var minSpaceSize = 0.5; // Multiplier for minimum space size
    var compact = true; // if true then try and fit as many words as possible. If false then try to
get the spacing as close as possible to normal
    var lineSpacing = 1.5; // space between lines
    const noJustifySetting = {  // This setting forces justified text off. Used to render last line
of paragraph.
        minSpaceSize : 1,
        maxSpaceSize : 1,
    }
```

```
    // Parse vet and set settings object.
    var justifiedTextSettings = function(settings){
        var min, max;
        var vetNumber = (num, defaultNum) => {
            num = num !== null && num !== null && !isNaN(num) ? num : defaultNum;
            return num < 0 ? defaultNum : num;
        }
        if(settings === undefined || settings === null){ return; }
        compact = settings.compact === true ? true : settings.compact === false ? false : compact;
        max = vetNumber(settings.maxSpaceSize, maxSpaceSize);
        min = vetNumber(settings.minSpaceSize, minSpaceSize);
        lineSpacing = vetNumber(settings.lineSpacing, lineSpacing);
        if(min > max){ return; }
        minSpaceSize = min;
        maxSpaceSize = max;
    }
    var getFontSize = function(font){  // get the font size.
        var numFind = /[0-9]+/;
        var number = numFind.exec(font)[0];
        if(isNaN(number)){
            throw new ReferenceError("justifiedPar Cant find font size");
        }
        return Number(number);
    }
    function justifiedPar(ctx, text, x, y, width, settings, stroke){
        var spaceWidth, minS, maxS, words, count, lines, lineWidth, lastLineWidth, lastSize, i,
renderer, fontSize, adjSpace, spaces, word, lineWords, lineFound;
        spaceWidth = ctx.measureText(" ").width;
        minS = spaceWidth * minSpaceSize;
        maxS = spaceWidth * maxSpaceSize;
        words = text.split(" ").map(word => {  // measure all words.
            var w = ctx.measureText(word).width;
            return {
                width : w,
                word : word,
            };
        });
        // count = num words, spaces = number spaces, spaceWidth normal space size
        // adjSpace new space size >= min size. useSize Resulting space size used to render
        count = 0;
        lines = [];
        // create lines by shifting words from the words array until the spacing is optimal. If
compact
        // true then will true and fit as many words as possible. Else it will try and get the
spacing as
        // close as possible to the normal spacing
        while(words.length > 0){
            lastLineWidth = 0;
            lastSize = -1;
            lineFound = false;
            // each line must have at least one word.
            word = words.shift();
            lineWidth = word.width;
            lineWords = [word.word];
            count = 0;
            while(lineWidth < width && words.length > 0){ // Add words to line
                word = words.shift();
                lineWidth += word.width;
                lineWords.push(word.word);
                count += 1;
                spaces = count - 1;
                adjSpace =  (width - lineWidth) / spaces;
```

```
                if(minS > adjSpace){  // if spacing less than min remove last word and finish line
                    lineFound = true;
                    words.unshift(word);
                    lineWords.pop();
                }else{
                    if(!compact){ // if compact mode
                        if(adjSpace < spaceWidth){ // if less than normal space width
                            if(lastSize === -1){
                                lastSize = adjSpace;
                            }
                            // check if with last word on if its closer to space width
                            if(Math.abs(spaceWidth - adjSpace) < Math.abs(spaceWidth - lastSize)){
                                lineFound = true; // yes keep it
                            }else{
                                words.unshift(word);  // no better fit if last word removes
                                lineWords.pop();
                                lineFound = true;
                            }
                        }
                    }
                }
                lastSize = adjSpace; // remember spacing
            }
            lines.push(lineWords.join(" ")); // and the line
        }
        // lines have been worked out get font size, render, and render all the lines. last
        // line may need to be rendered as normal so it is outside the loop.
        fontSize = getFontSize(ctx.font);
        renderer = stroke === true ? ctx.strokeJustifyText.bind(ctx) :
ctx.fillJustifyText.bind(ctx);
        for(i = 0; i < lines.length - 1; i ++){
            renderer(lines[i], x, y, width, settings);
            y += lineSpacing * fontSize;
        }
        if(lines.length > 0){ // last line if left or start aligned for no justify
            if(ctx.textAlign === "left" || ctx.textAlign === "start"){
                renderer(lines[lines.length - 1], x, y, width, noJustifySetting);
                ctx.measureJustifiedText("", width, settings);
            }else{
                renderer(lines[lines.length - 1], x, y, width);
            }
        }
        // return details about the paragraph.
        y += lineSpacing * fontSize;
        return {
            nextLine : y,
            fontSize : fontSize,
            lineHeight : lineSpacing * fontSize,
        };
    }
    // define fill
    var fillParagraphText = function(text, x, y, width, settings){
        justifiedTextSettings(settings);
        settings = {
            minSpaceSize : minSpaceSize,
            maxSpaceSize : maxSpaceSize,
        };
        return justifiedPar(this, text, x, y, width, settings);
    }
    // define stroke
    var strokeParagraphText = function(text, x, y, width, settings){
        justifiedTextSettings(settings);
```

```
        settings = {
            minSpaceSize : minSpaceSize,
            maxSpaceSize : maxSpaceSize,
        };
        return justifiedPar(this, text, x, y, width, settings,true);
    }
    CanvasRenderingContext2D.prototype.fillParaText = fillParagraphText;
    CanvasRenderingContext2D.prototype.strokeParaText = strokeParagraphText;
})();
```

> **NOTE** this extends the `CanvasRenderingContext2D` prototype. If you do not wish this to happen use the example **Justified text** to work out how to change this example to be part of the global namespace.

> **NOTE** Will throw a ReferenceError if this example can not find the function `CanvasRenderingContext2D.prototype.fillJustifyText`

How to use

```
ctx.fillParaText(text, x, y, width, [settings]);
ctx.strokeParaText(text, x, y, width, [settings]);
```

See **Justified text** for details on arguments. Arguments between `[` and `]` are optional.

The `settings` argument has two additional properties.

- **compact:** Default `true`. If true tries to pack as many words as possible per line. If false the tries to get the word spacing as close as possible to normal spacing.
- **lineSpacing** Default `1.5`. Space per line default `1.5` the distance from on line to the next in terms of font size

Properties missing from the settings object will default to their default values or to the last valid values. The properties will only be changed if the new values are valid. For `compact` valid values are only booleans `true` or `false` Truthy values are not considered valid.

Return object

The two functions return an object containing information to help you place the next paragraph. The object contains the following properties.

- **nextLine** Position of the next line after the paragraph pixels.
- **fontSize** Size of the font. (please note only use fonts defined in pixels eg `14px arial`)
- **lineHeight** Distance in pixels from one line to the next

This example uses a simple algorithm that works one line at to time to find the best fit for a paragraph. This does not mean that it the best fit (rather the algorithm's best) You may wish to improve the algorithm by creating a multi pass line algorithm over the generated lines. Moving words from the end of one line to the start of the next, or from the start back to the end. The best look is achieved when the spacing over the entire paragraph has the smallest variation and is the closest to the normal text spacing.

As this example is dependent on the **Justified text** example the code is very similar. You may wish to move the two into one function. Replace the function `justifiedTextSettings` in the other example with the one used in this example. Then copy all the rest of the code from this example into the anonymous function body of the **Justified text** example. You will no longer need to test for dependencies found at `// Code point A` It can be removed.

Usage example

```
ctx.font = "25px arial";
ctx.textAlign = "center"

var left = 10;
var center = canvas.width / 2;
var width = canvas.width-left*2;
var y = 20;
var size = 16;
var i = 0;
ctx.fillText("Justified paragraph examples.",center,y);
y+= 30;
ctx.font = "14px arial";
ctx.textAlign = "left"
// set para settings
var setting = {
    maxSpaceSize : 6,
    minSpaceSize : 0.5,
    lineSpacing : 1.2,
    compact : true,
}
// Show the left and right bounds.
ctx.strokeStyle = "red"
ctx.beginPath();
ctx.moveTo(left,y - size * 2);
ctx.lineTo(left, y + size * 15);
ctx.moveTo(canvas.width - left,y - size * 2);
ctx.lineTo(canvas.width - left, y + size * 15);
ctx.stroke();
ctx.textAlign = "left";
ctx.fillStyle = "black";

// Draw paragraph
var line = ctx.fillParaText(para, left, y, width, setting);  // settings is remembered

// Next paragraph
y = line.nextLine + line.lineHeight;
setting.compact = false;
ctx.fillParaText(para, left, y, width, setting);
```

> **Note:** For text aligned `left` or `start` the last line of tha paragraph will always have normal spacing. For all other alignments the last line is treated like all others.

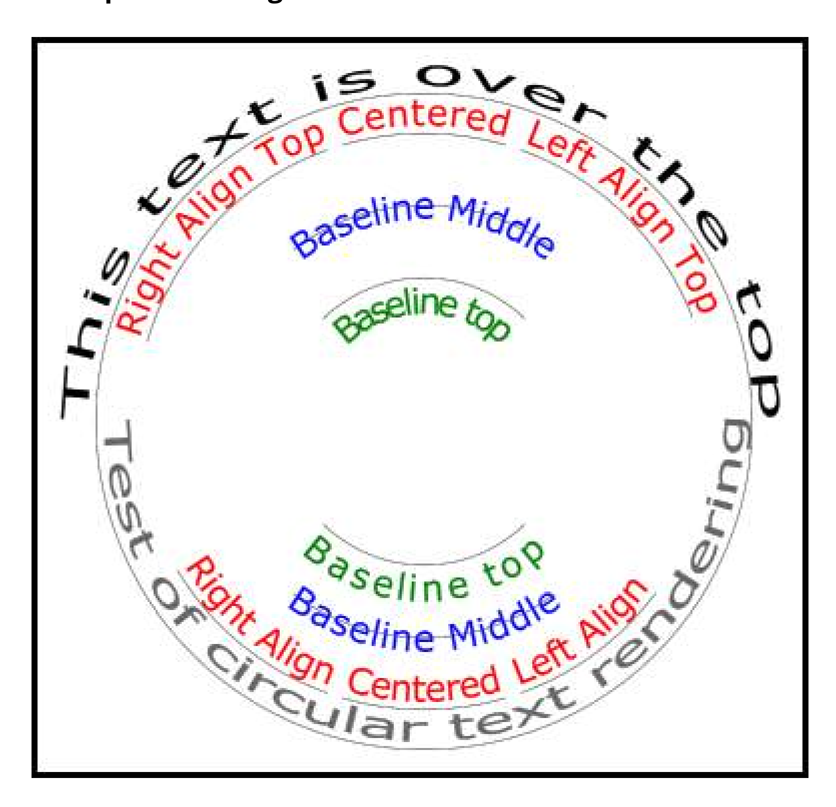
> **Note:** You can inset the start of the paragraph with spaces. Though this may not be consistent from paragraph to paragraph. It is always a good thing to learn what a function is doing and modifying it. An exercise would be to add a setting to the settings that indents the first line by a fixed amount. Hint the while loop will need to temporarily make the first word appear larger (+ indent) `words[0].width += ?` and then when rendering lines indent the first line.

Section 2.3: Rendering text along an arc

This example shows how to render text along an arc. It includes how you can add functionality to the `CanvasRenderingContext2D` by extending its prototype.

This examples is derived from the stackoverflow answer Circular Text.

Example rendering



Example code

The example adds 3 new text rendering functions to the 2D context prototype.

- **ctx.fillCircleText(text, x, y, radius, start, end, forward);**
- **ctx.strokeCircleText(text, x, y, radius, start, end, forward);**
- **ctx.measureCircleText(text, radius);**

```
(function(){
    const FILL = 0;        // const to indicate filltext render
    const STROKE = 1;
    var renderType = FILL; // used internal to set fill or stroke text
    const multiplyCurrentTransform = true; // if true Use current transform when rendering
                                           // if false use absolute coordinates which is a little
quicker
                                           // after render the currentTransform is restored to
default transform



    // measure circle text
    // ctx: canvas context
    // text: string of text to measure
    // r: radius in pixels
    //
    // returns the size metrics of the text
    //
    // width: Pixel width of text
    // angularWidth : angular width of text in radians
    // pixelAngularSize : angular width of a pixel in radians
    var measure = function(ctx, text, radius){
        var textWidth = ctx.measureText(text).width; // get the width of all the text
        return {
            width               : textWidth,
```

```
            angularWidth        : (1 / radius) * textWidth,
            pixelAngularSize    : 1 / radius
        };
    }

    // displays text along a circle
    // ctx: canvas context
    // text: string of text to measure
    // x,y: position of circle center
    // r: radius of circle in pixels
    // start: angle in radians to start.
    // [end]: optional. If included text align is ignored and the text is
    //        scaled to fit between start and end;
    // [forward]: optional default true. if true text direction is forwards, if false  direction is
backward
    var circleText = function (ctx, text, x, y, radius, start, end, forward) {
        var i, textWidth, pA, pAS, a, aw, wScale, aligned, dir, fontSize;
        if(text.trim() === "" || ctx.globalAlpha === 0){ // don't render empty string or
transparent
            return;
        }
        if(isNaN(x) || isNaN(y) || isNaN(radius) || isNaN(start) || (end !== undefined && end !==
null && isNaN(end))){ //
            throw TypeError("circle text arguments requires a number for x,y, radius, start, and
end.")
        }
        aligned = ctx.textAlign;        // save the current textAlign so that it can be restored at
end
        dir = forward ? 1 : forward === false ? -1 : 1;  // set dir if not true or false set
forward as true
        pAS = 1 / radius;               // get the angular size of a pixel in radians
        textWidth = ctx.measureText(text).width; // get the width of all the text
        if (end !== undefined && end !== null) { // if end is supplied then fit text between start
and end
            pA = ((end - start) / textWidth) * dir;
            wScale = (pA / pAS) * dir;
        } else {                 // if no end is supplied correct start and end for alignment
            // if forward is not given then swap top of circle text to read the correct direction
            if(forward === null || forward === undefined){
                if(((start % (Math.PI * 2)) + Math.PI * 2) % (Math.PI * 2) > Math.PI){
                    dir = -1;
                }
            }
            pA = -pAS * dir ;
            wScale = -1 * dir;
            switch (aligned) {
            case "center":       // if centered move around half width
                start -= (pA * textWidth )/2;
                end = start + pA * textWidth;
                break;
            case "right":// intentionally falls through to case "end"
            case "end":
                end = start;
                start -= pA * textWidth;
                break;
            case "left":  // intentionally falls through to case "start"
            case "start":
                end = start + pA * textWidth;
            }
        }

        ctx.textAlign = "center";                     // align for rendering
```

```
        a = start;                                    // set the start angle
        for (var i = 0; i < text.length; i += 1) {    // for each character
            aw = ctx.measureText(text[i]).width * pA; // get the angular width of the text
            var xDx = Math.cos(a + aw / 2);           // get the yAxies vector from the center x,y
out
            var xDy = Math.sin(a + aw / 2);
            if(multiplyCurrentTransform){ // transform multiplying current transform
                ctx.save();
                if (xDy < 0) { // is the text upside down. If it is flip it
                    ctx.transform(-xDy * wScale, xDx * wScale, -xDx, -xDy, xDx * radius + x, xDy *
radius + y);
                } else {
                    ctx.transform(-xDy * wScale, xDx * wScale, xDx, xDy, xDx * radius + x, xDy *
radius + y);
                }
            }else{
                if (xDy < 0) { // is the text upside down. If it is flip it
                    ctx.setTransform(-xDy * wScale, xDx * wScale, -xDx, -xDy, xDx * radius + x, xDy
* radius + y);
                } else {
                    ctx.setTransform(-xDy * wScale, xDx * wScale, xDx, xDy, xDx * radius + x, xDy *
radius + y);
                }
            }
            if(renderType === FILL){
                ctx.fillText(text[i], 0, 0);    // render the character
            }else{
                ctx.strokeText(text[i], 0, 0);  // render the character
            }
            if(multiplyCurrentTransform){  // restore current transform
                ctx.restore();
            }
            a += aw;                     // step to the next angle
        }
        // all done clean up.
        if(!multiplyCurrentTransform){
            ctx.setTransform(1, 0, 0, 1, 0, 0); // restore the transform
        }
        ctx.textAlign = aligned;            // restore the text alignment
    }
    // define fill text
    var fillCircleText = function(text, x, y, radius, start, end, forward){
        renderType = FILL;
        circleText(this, text, x, y, radius, start, end, forward);
    }
    // define stroke text
    var strokeCircleText = function(text, x, y, radius, start, end, forward){
        renderType = STROKE;
        circleText(this, text, x, y, radius, start, end, forward);
    }
    // define measure text
    var measureCircleTextExt = function(text,radius){
        return measure(this, text, radius);
    }
    // set the prototypes
    CanvasRenderingContext2D.prototype.fillCircleText = fillCircleText;
    CanvasRenderingContext2D.prototype.strokeCircleText = strokeCircleText;
    CanvasRenderingContext2D.prototype.measureCircleText = measureCircleTextExt;
})();
```

Function descriptions

This example adds 3 functions to the `CanvasRenderingContext2D prototype`. `fillCircleText`, `strokeCircleText`, and `measureCircleText`

CanvasRenderingContext2D.fillCircleText(text, x, y, radius, start, [end, [forward]]);

CanvasRenderingContext2D.strokeCircleText(text, x, y, radius, start, [end, [forward]]);

- **text:** Text to render as String.
- **x**,**y**: Position of circle center as Numbers.
- **radius:** radius of circle in pixels
- **start:** angle in radians to start.
- **[end]:** optional. If included `ctx.textAlign` is ignored and the text is scaled to fit between start and end.
- **[forward]:** optional default 'true'. if true text direction is forwards, if 'false' direction is backward.

Both functions use the textBaseline to position the text vertically around the radius. For the best results use `ctx.TextBaseline`.

Functions will throw a `TypeError` is any of the numerical arguments as NaN.

If the `text` argument trims to an empty string or `ctx.globalAlpha = 0` the function just drops through and does nothing.

CanvasRenderingContext2D.measureCircleText(text, radius);

```
- **text:** String of text to measure.
- **radius:** radius of circle in pixels.
```

Returns a Object containing various size metrics for rendering circular text

```
  - **width:** Pixel width of text as it would normaly be rendered
  - **angularWidth:** angular width of text in radians.
  - **pixelAngularSize:** angular width of a pixel in radians.
```

Usage examples

```
const rad = canvas.height * 0.4;
const text = "Hello circle TEXT!";
const fontSize = 40;
const centX = canvas.width / 2;
const centY = canvas.height / 2;
ctx.clearRect(0,0,canvas.width,canvas.height)

ctx.font = fontSize + "px verdana";
ctx.textAlign = "center";
ctx.textBaseline = "bottom";
ctx.fillStyle = "#000";
ctx.strokeStyle = "#666";

// Text under stretched from Math.PI to 0 (180 - 0 deg)
ctx.fillCircleText(text, centX, centY, rad, Math.PI, 0);

// text over top centered at Math.PI * 1.5 ( 270 deg)
ctx.fillCircleText(text, centX, centY, rad, Math.PI * 1.5);

// text under top centered at Math.PI * 1.5 ( 270 deg)
ctx.textBaseline = "top";
ctx.fillCircleText(text, centX, centY, rad, Math.PI * 1.5);
```

```
// text over top centered at Math.PI * 1.5 ( 270 deg)
ctx.textBaseline = "middle";
ctx.fillCircleText(text, centX, centY, rad, Math.PI * 1.5);


// Use measureCircleText to get angular size
var circleTextMetric = ctx.measureCircleText("Text to measure", rad);
console.log(circleTextMetric.width);            // width of text if rendered normally
console.log(circleTextMetric.angularWidth);     // angular width of text
console.log(circleTextMetric.pixelAngularSize); // angular size of a pixel



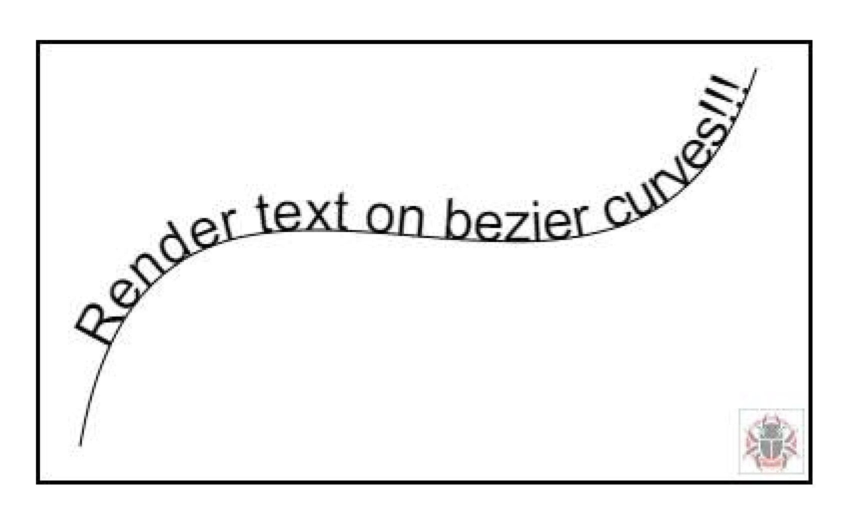
// Use measure text to draw a arc around the text
ctx.textBaseline = "middle";
var width = ctx.measureCircleText(text, rad).angularWidth;
ctx.fillCircleText(text, centX, centY, rad, Math.PI * 1.5);

// render the arc around the text
ctx.strokeStyle= "red";
ctx.lineWidth = 3;
ctx.beginPath();
ctx.arc(centX, centY, rad + fontSize / 2,Math.PI * 1.5 - width/2,Math.PI*1.5 + width/2);
ctx.arc(centX, centY, rad - fontSize / 2,Math.PI * 1.5 + width/2,Math.PI*1.5 - width/2,true);
ctx.closePath();
ctx.stroke();
```

> **NOTE:** The text rendered is only an approximation of circular text. For example if two l's are rendered the two lines will not be parallel, but if you render a "H" the two edges will be parallel. This is because each character is rendered as close as possible to the required direction, rather than each pixel being correctly transformed to create circular text.

> **NOTE:** `const multiplyCurrentTransform = true;` defined in this example is used to set the transformation method used. If `false` the transformation for circular text rendering is absolute and does not depend on the current transformation state. The text will not be effected by any previous scale, rotate, or translate transforms. This will increase the performance of the render function, after the function is called the transform will be set to the default `setTransform(1,0,0,1,0,0)`
>
> If `multiplyCurrentTransform = true` (set as default in this example) the text will use the current transform so that the text can be scaled translated, skewed, rotated, etc but modifying the current transform befor calling the `fillCircleText` and `strokeCircleText` functions. Depending on the current state of the 2D context this may be somewhat slower then `multiplyCurrentTransform = false`

Section 2.4: Text on curve, cubic and quadratic beziers



textOnCurve(text,offset,x1,y1,x2,y2,x3,y3,x4,y4)

Renders text on quadratic and cubic curves.

- `text` is the text to render
- `offset` distance from start of curve to text >= 0
- `x1,y1` - `x3,y3` points of quadratic curve or
- `x1,y1` - `x4,y4` points of cubic curve or

Example usage:

```
textOnCurve("Hello world!",50,100,100,200,200,300,100); // draws text on quadratic curve
                                                        // 50 pixels from start of curve


textOnCurve("Hello world!",50,100,100,200,200,300,100,400,200);
                                                        // draws text on cubic curve
                                                        // 50 pixels from start of curve
```

The Function and curver helper function

```
// pass 8 values for cubic bezier
// pass 6 values for quadratic
// Renders text from start of curve
var textOnCurve = function(text,offset,x1,y1,x2,y2,x3,y3,x4,y4){
    ctx.save();
    ctx.textAlign = "center";
    var widths = [];
    for(var i = 0; i < text.length; i ++){
        widths[widths.length] = ctx.measureText(text[i]).width;
    }
    var ch = curveHelper(x1,y1,x2,y2,x3,y3,x4,y4);
    var pos = offset;
    var cpos = 0;

    for(var i = 0; i < text.length; i ++){
        pos += widths[i] / 2;
        cpos = ch.forward(pos);
        ch.tangent(cpos);
        ctx.setTransform(ch.vect.x, ch.vect.y, -ch.vect.y, ch.vect.x, ch.vec.x, ch.vec.y);
        ctx.fillText(text[i],0,0);

        pos += widths[i] / 2;
    }
    ctx.restore();
}
```

The curve helper function is designed to increase the performance of finding points on the bezier.

```
// helper function locates points on bezier curves.
function curveHelper(x1, y1, x2, y2, x3, y3, x4, y4){
    var tx1, ty1, tx2, ty2, tx3, ty3, tx4, ty4;
    var a,b,c,u;
    var vec,currentPos,vec1,vect;
    vec = {x:0,y:0};
    vec1 = {x:0,y:0};
    vect = {x:0,y:0};
    quad = false;
    currentPos = 0;
    currentDist = 0;
    if(x4 === undefined || x4 === null){
        quad = true;
        x4 = x3;
        y4 = y3;
    }
    var estLen = Math.sqrt((x4 - x1) * (x4 - x1) + (y4 - y1) * (y4 - y1));
    var onePix = 1 / estLen;
    function posAtC(c){
        tx1 = x1; ty1 = y1;
        tx2 = x2; ty2 = y2;
        tx3 = x3; ty3 = y3;
        tx1 += (tx2 - tx1) * c;
        ty1 += (ty2 - ty1) * c;
        tx2 += (tx3 - tx2) * c;
        ty2 += (ty3 - ty2) * c;
        tx3 += (x4 - tx3) * c;
        ty3 += (y4 - ty3) * c;
        tx1 += (tx2 - tx1) * c;
        ty1 += (ty2 - ty1) * c;
        tx2 += (tx3 - tx2) * c;
        ty2 += (ty3 - ty2) * c;
        vec.x = tx1 + (tx2 - tx1) * c;
        vec.y = ty1 + (ty2 - ty1) * c;
        return vec;
    }
    function posAtQ(c){
        tx1 = x1; ty1 = y1;
        tx2 = x2; ty2 = y2;
        tx1 += (tx2 - tx1) * c;
        ty1 += (ty2 - ty1) * c;
        tx2 += (x3 - tx2) * c;
        ty2 += (y3 - ty2) * c;
        vec.x = tx1 + (tx2 - tx1) * c;
        vec.y = ty1 + (ty2 - ty1) * c;
        return vec;
    }
    function forward(dist){
        var step;
        helper.posAt(currentPos);

        while(currentDist < dist){
            vec1.x = vec.x;
            vec1.y = vec.y;
            currentPos += onePix;
            helper.posAt(currentPos);
            currentDist += step = Math.sqrt((vec.x - vec1.x) * (vec.x - vec1.x) + (vec.y - vec1.y)
* (vec.y - vec1.y));

        }
```

```
        currentPos -= ((currentDist - dist) / step) * onePix
        currentDist -= step;
        helper.posAt(currentPos);
        currentDist += Math.sqrt((vec.x - vec1.x) * (vec.x - vec1.x) + (vec.y - vec1.y) * (vec.y -
vec1.y));
        return currentPos;
    }

    function tangentQ(pos){
        a = (1-pos) * 2;
        b = pos * 2;
        vect.x = a * (x2 - x1) + b * (x3 - x2);
        vect.y = a * (y2 - y1) + b * (y3 - y2);
        u = Math.sqrt(vect.x * vect.x + vect.y * vect.y);
        vect.x /= u;
        vect.y /= u;
    }
    function tangentC(pos){
        a  = (1-pos)
        b  = 6 * a * pos;
        a *= 3 * a;
        c  = 3 * pos * pos;
        vect.x  = -x1 * a + x2 * (a - b) + x3 * (b - c) + x4 * c;
        vect.y  = -y1 * a + y2 * (a - b) + y3 * (b - c) + y4 * c;
        u = Math.sqrt(vect.x * vect.x + vect.y * vect.y);
        vect.x /= u;
        vect.y /= u;
    }
    var helper = {
        vec : vec,
        vect : vect,
        forward : forward,
    }
    if(quad){
        helper.posAt = posAtQ;
        helper.tangent = tangentQ;
    }else{
        helper.posAt = posAtC;
        helper.tangent = tangentC;
    }
    return helper
}
```

Section 2.5: Drawing Text

Drawing to canvas isn't just limited to shapes and images. You can also draw text to the canvas.

To draw text on the canvas, get a reference to the canvas and then call the `fillText` method on the context.

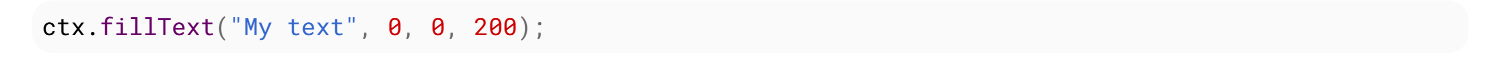
```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');
ctx.fillText("My text", 0, 0);
```

The three **required** arguments that are passed into `fillText` are:

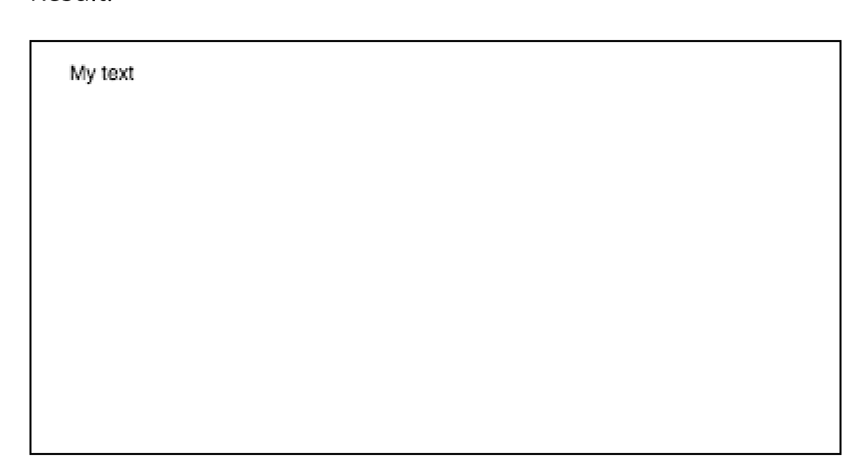
1. The text that you would like to display
2. The horizontal (x-axis) position
3. The vertical (y-axis) position

Additionally, there is a fourth **optional** argument, which you can use to specify the maximum width of your text in


pixels. In the example below the value of `200` restricts the maximum width of the text to 200px:


```
ctx.fillText("My text", 0, 0, 200);
```

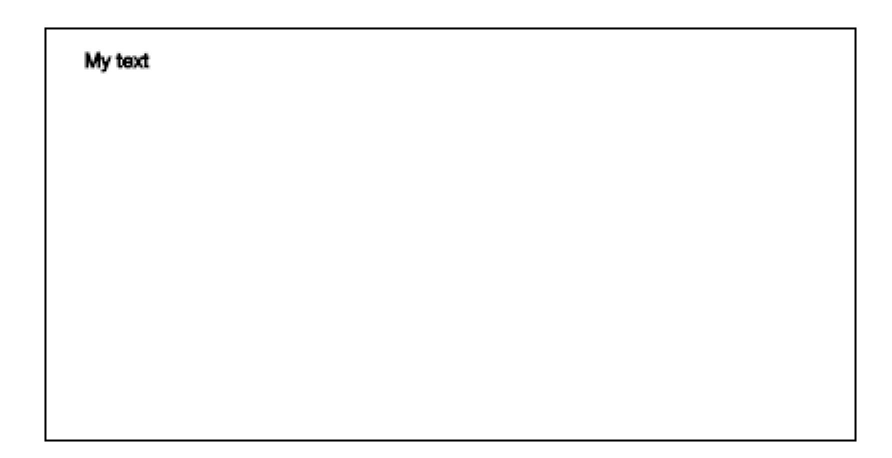
Result:



You can also draw text without a fill, and just an outline instead, using the `strokeText` method:

```
ctx.strokeText("My text", 0, 0);
```

Result:



Without any font formatting properties applied, the canvas renders text at 10px in sans-serif by default, making it hard to see the difference between the result of the `fillText` and `strokeText` methods. See the Formatting Text example for details on how to increase text size and apply other aesthetic changes to text.

Section 2.6: Formatting Text

The default font formatting provided by the `fillText` and `strokeText` methods isn't very aesthetically appealing. Fortunately the canvas API provides properties for formatting text.

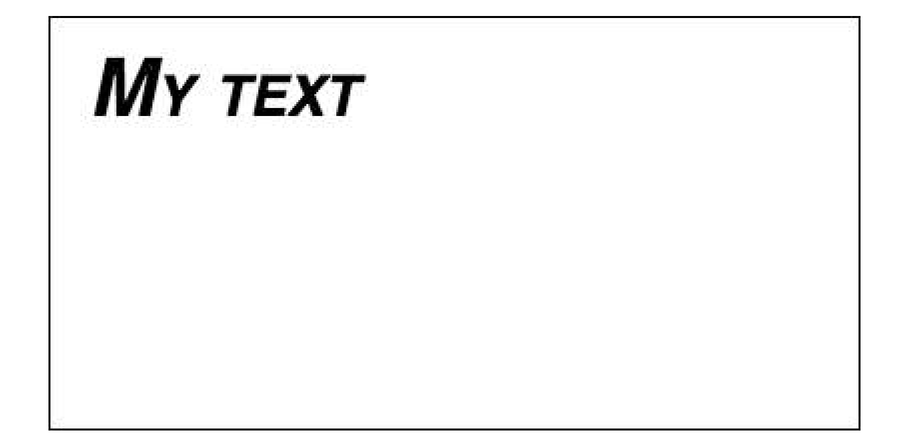
Using the `font` property you can specify:

- font-style
- font-variant
- font-weight
- font-size / line-height
- font-family

For example:

```
ctx.font = "italic small-caps bold 40px Helvetica, Arial, sans-serif";
ctx.fillText("My text", 20, 50);
```

Result:



Using the `textAlign` property you can also change text alignment to either:

- left
- center
- right
- end (same as right)
- start (same as left)

For example:

```
ctx.textAlign = "center";
```

Section 2.7: Wrapping text into paragraphs

Native Canvas API does not have a method to wrap text onto the next line when a desired maximum width is reached. This example wraps text into paragraphs.

```
function wrapText(text, x, y, maxWidth, fontSize, fontFace){
  var firstY=y;
  var words = text.split(' ');
  var line = '';
  var lineHeight=fontSize*1.286; // a good approx for 10-18px sizes

  ctx.font=fontSize+" "+fontFace;
  ctx.textBaseline='top';

  for(var n = 0; n < words.length; n++) {
    var testLine = line + words[n] + ' ';
    var metrics = ctx.measureText(testLine);
    var testWidth = metrics.width;
    if(testWidth > maxWidth) {
      ctx.fillText(line, x, y);
      if(n<words.length-1){
          line = words[n] + ' ';
          y += lineHeight;
      }
    }
    else {
      line = testLine;
    }
```

```
  }
  ctx.fillText(line, x, y);
}
```

Section 2.8: Draw text paragraphs into irregular shapes

This example draws text paragraphs into any portions of the canvas that have opaque pixels.

It works by finding the next block of opaque pixels that is large enough to contain the next specified word and filling that block with the specified word.

The opaque pixels can come from any source: Path drawing commands and /or images.

```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; padding:10px; }
    #canvas{border:1px solid red;}
</style>
<script>
window.onload=(function(){

    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    var cw=canvas.width;
    var ch=canvas.height;

    var fontsize=12;
    var fontface='verdana';
    var lineHeight=parseInt(fontsize*1.286);

    var text='It was the best of times, it was the worst of times, it was the age of wisdom, it was
the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the
```

```
season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of
despair, we had everything before us, we had nothing before us, we were all going direct to Heaven,
we were all going direct the other way; in short, the period was so far like the present period,
that some of its noisiest authorities insisted on its being received, for good or for evil, in the
superlative degree of comparison only.';
    var words=text.split(' ');
    var wordWidths=[];
    ctx.font=fontsize+'px '+fontface;
    for(var i=0;i<words.length;i++){ wordWidths.push(ctx.measureText(words[i]).width); }
    var spaceWidth=ctx.measureText(' ').width;
    var wordIndex=0
    var data=[];

    // Demo: draw Heart
    // Note: the shape can be ANY opaque drawing -- even an image
    ctx.scale(3,3);
    ctx.beginPath();
    ctx.moveTo(75,40);
    ctx.bezierCurveTo(75,37,70,25,50,25);
    ctx.bezierCurveTo(20,25,20,62.5,20,62.5);
    ctx.bezierCurveTo(20,80,40,102,75,120);
    ctx.bezierCurveTo(110,102,130,80,130,62.5);
    ctx.bezierCurveTo(130,62.5,130,25,100,25);
    ctx.bezierCurveTo(85,25,75,37,75,40);
    ctx.fillStyle='red';
    ctx.fill();
    ctx.setTransform(1,0,0,1,0,0);

    // fill heart with text
    ctx.fillStyle='white';
    var imgDataData=ctx.getImageData(0,0,cw,ch).data;
    for(var i=0;i<imgDataData.length;i+=4){
        data.push(imgDataData[i+3]);
    }
    placeWords();

    // draw words sequentially into next available block of
    //    available opaque pixels
    function placeWords(){
        var sx=0;
        var sy=0;
        var y=0;
        var wordIndex=0;
        ctx.textBaseline='top';
        while(y<ch && wordIndex<words.length){
            sx=0;
            sy=y;
            var startingIndex=wordIndex;
            while(sx<cw && wordIndex<words.length){
                var x=getRect(sx,sy,lineHeight);
                var available=x-sx;
                var spacer=spaceWidth;  // spacer=0 to have no left margin
                var w=spacer+wordWidths[wordIndex];
                while(available>=w){
                    ctx.fillText(words[wordIndex],spacer+sx,sy);
                    sx+=w;
                    available-=w;
                    spacer=spaceWidth;
                    wordIndex++;
                    w=spacer+wordWidths[wordIndex];
                }
                sx=x+1;
```

```
            }
            y=(wordIndex>startingIndex)?y+lineHeight:y+1;
        }
    }

    // find a rectangular block of opaque pixels
    function getRect(sx,sy,height){
        var x=sx;
        var y=sy;
        var ok=true;
        while(ok){
            if(data[y*cw+x]<250){ok=false;}
            y++;
            if(y>=sy+height){
                y=sy;
                x++;
                if(x>=cw){ok=false;}
            }
        }
        return(x);
    }

}); // end $(function(){});
</script>
</head>
<body>
    <h4>Note: the shape must be closed and alpha>=250 inside</h4>
    <canvas id="canvas" width=400 height=400></canvas>
</body>
</html>
```

Section 2.9: Fill text with an image

This example fills text with a specified image.

Important! The specified image must be fully loaded before calling this function or the drawing will fail. Use `image.onload` to be sure the image is fully loaded.

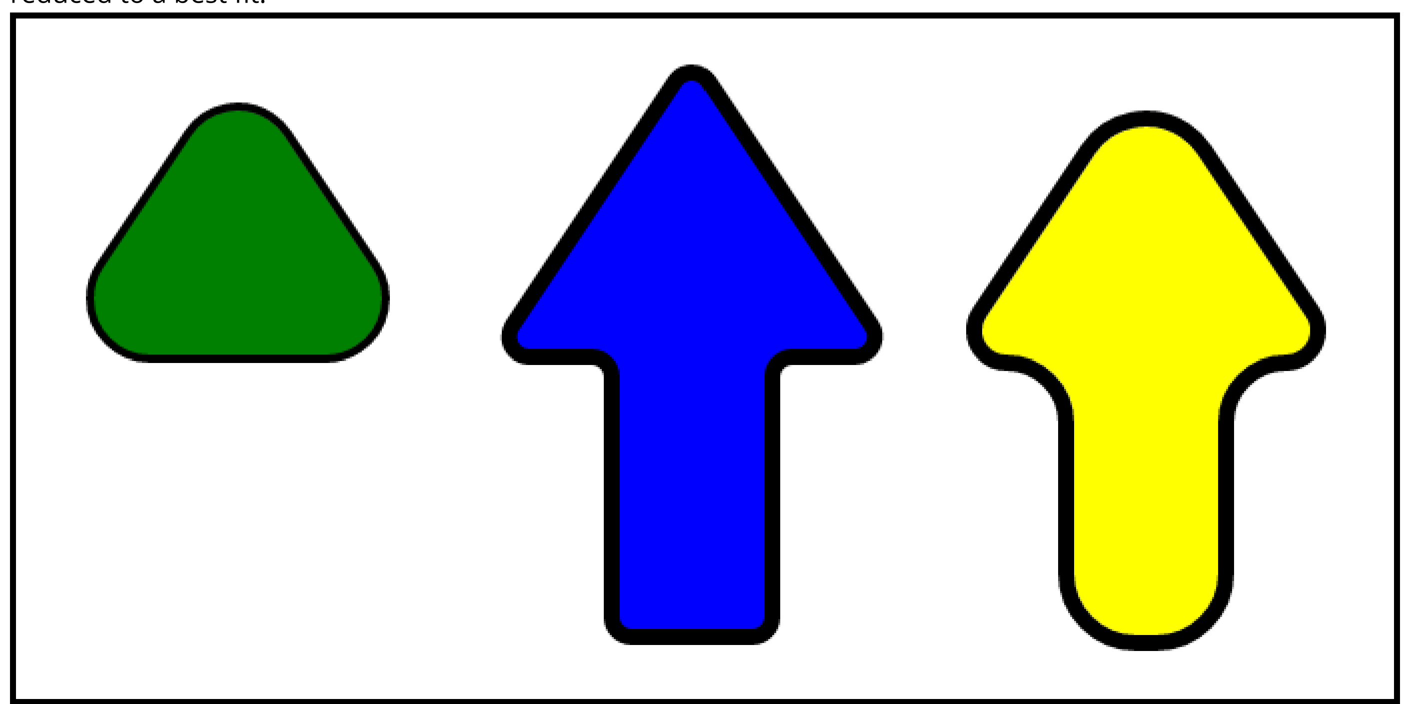



```
function drawImageInsideText(canvas,x,y,img,text,font){
    var c=canvas.cloneNode();
    var ctx=c.getContext('2d');
    ctx.font=font;
    ctx.fillText(text,x,y);
    ctx.globalCompositeOperation='source-atop';
    ctx.drawImage(img,0,0);
    canvas.getContext('2d').drawImage(c,0,0);
}
```

Chapter 3: Polygons

Section 3.1: Render a rounded polygon

Creates a path from a set of points `[{x:?,y:?},{x:?,y:?},...,{x:?,y:?}]` with rounded corners of radius. If the corner angle is too small to fit the radius or the distance between corners does not allow room the corners radius is reduced to a best fit.



Usage Example

```
var triangle = [
    { x: 200, y : 50 },
    { x: 300, y : 200 },
    { x: 100, y : 200 }
];
var cornerRadius = 30;
ctx.lineWidth = 4;
ctx.fillStyle = "Green";
ctx.strokeStyle = "black";
ctx.beginPath(); // start a new path
roundedPoly(triangle, cornerRadius);
ctx.fill();
ctx.stroke();
```

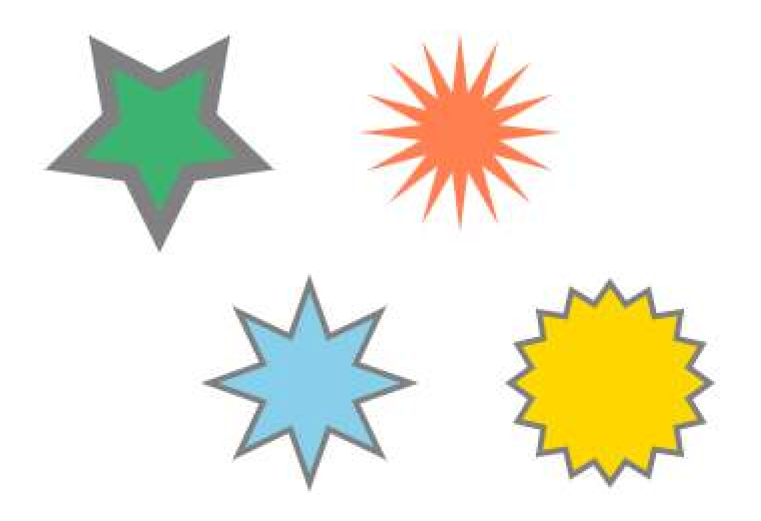
Render function

```
var roundedPoly = function(points,radius){
    var i, x, y, len, p1, p2, p3, v1, v2, sinA, sinA90, radDirection, drawDirection, angle,
halfAngle, cRadius, lenOut;
    var asVec = function (p, pp, v) { // convert points to a line with len and normalised
        v.x = pp.x - p.x; // x,y as vec
        v.y = pp.y - p.y;
        v.len = Math.sqrt(v.x * v.x + v.y * v.y); // length of vec
        v.nx = v.x / v.len; // normalised
        v.ny = v.y / v.len;
        v.ang = Math.atan2(v.ny, v.nx); // direction of vec
    }
    v1 = {};
```

```
    v2 = {};
    len = points.length;                          // number points
    p1 = points[len - 1];                         // start at end of path
    for (i = 0; i < len; i++) {                   // do each corner
        p2 = points[(i) % len];                   // the corner point that is being rounded
        p3 = points[(i + 1) % len];
        // get the corner as vectors out away from corner
        asVec(p2, p1, v1);                        // vec back from corner point
        asVec(p2, p3, v2);                        // vec forward from corner point
        // get corners cross product (asin of angle)
        sinA = v1.nx * v2.ny - v1.ny * v2.nx;     // cross product
        // get cross product of first line and perpendicular second line
        sinA90 = v1.nx * v2.nx - v1.ny * -v2.ny; // cross product to normal of line 2
        angle = Math.asin(sinA);                  // get the angle
        radDirection = 1;                         // may need to reverse the radius
        drawDirection = false;                    // may need to draw the arc anticlockwise
        // find the correct quadrant for circle center
        if (sinA90 < 0) {
            if (angle < 0) {
                angle = Math.PI + angle; // add 180 to move us to the 3 quadrant
            } else {
                angle = Math.PI - angle; // move back into the 2nd quadrant
                radDirection = -1;
                drawDirection = true;
            }
        } else {
            if (angle > 0) {
                radDirection = -1;
                drawDirection = true;
            }
        }
        halfAngle = angle / 2;
        // get distance from corner to point where round corner touches line
        lenOut = Math.abs(Math.cos(halfAngle) * radius / Math.sin(halfAngle));
        if (lenOut > Math.min(v1.len / 2, v2.len / 2)) { // fix if longer than half line length
            lenOut = Math.min(v1.len / 2, v2.len / 2);
            // ajust the radius of corner rounding to fit
            cRadius = Math.abs(lenOut * Math.sin(halfAngle) / Math.cos(halfAngle));
        } else {
            cRadius = radius;
        }
        x = p2.x + v2.nx * lenOut; // move out from corner along second line to point where rounded
circle touches
        y = p2.y + v2.ny * lenOut;
        x += -v2.ny * cRadius * radDirection; // move away from line to circle center
        y += v2.nx * cRadius * radDirection;
        // x,y is the rounded corner circle center
        ctx.arc(x, y, cRadius, v1.ang + Math.PI / 2 * radDirection, v2.ang - Math.PI / 2 *
radDirection, drawDirection); // draw the arc clockwise
        p1 = p2;
        p2 = p3;
    }
    ctx.closePath();
}
```

Section 3.2: Stars

Draw stars with flexible styling (size, colors, number-of-points).



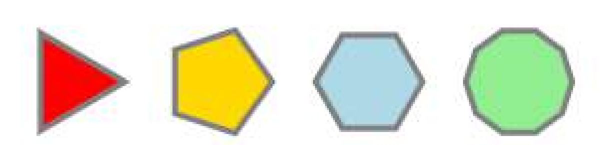
```
// Usage:
drawStar(75,75,5,50,25,'mediumseagreen','gray',9);
drawStar(150,200,8,50,25,'skyblue','gray',3);
drawStar(225,75,16,50,20,'coral','transparent',0);
drawStar(300,200,16,50,40,'gold','gray',3);

// centerX, centerY: the center point of the star
// points: the number of points on the exterior of the star
// inner: the radius of the inner points of the star
// outer: the radius of the outer points of the star
// fill, stroke: the fill and stroke colors to apply
// line: the linewidth of the stroke

function drawStar(centerX, centerY, points, outer, inner, fill, stroke, line) {
    // define the star
    ctx.beginPath();
    ctx.moveTo(centerX, centerY+outer);
    for (var i=0; i < 2*points+1; i++) {
        var r = (i%2 == 0)? outer : inner;
        var a = Math.PI * i/points;
        ctx.lineTo(centerX + r*Math.sin(a), centerY + r*Math.cos(a));
    };
    ctx.closePath();
    // draw
    ctx.fillStyle=fill;
    ctx.fill();
    ctx.strokeStyle=stroke;
    ctx.lineWidth=line;
    ctx.stroke()
}
```

Section 3.3: Regular Polygon

A regular polygon has all sides equal length.



```
// Usage:
drawRegularPolygon(3,25,75,50,6,'gray','red',0);
drawRegularPolygon(5,25,150,50,6,'gray','gold',0);
```

```
drawRegularPolygon(6,25,225,50,6,'gray','lightblue',0);
drawRegularPolygon(10,25,300,50,6,'gray','lightgreen',0);

function
drawRegularPolygon(sideCount,radius,centerX,centerY,strokeWidth,strokeColor,fillColor,rotationRadia
ns){
    var angles=Math.PI*2/sideCount;
    ctx.translate(centerX,centerY);
    ctx.rotate(rotationRadians);
    ctx.beginPath();
    ctx.moveTo(radius,0);
    for(var i=1;i<sideCount;i++){
        ctx.rotate(angles);
        ctx.lineTo(radius,0);
    }
    ctx.closePath();
    ctx.fillStyle=fillColor;
    ctx.strokeStyle = strokeColor;
    ctx.lineWidth = strokeWidth;
    ctx.stroke();
    ctx.fill();
    ctx.rotate(angles*-(sideCount-1));
    ctx.rotate(-rotationRadians);
    ctx.translate(-centerX,-centerY);
}
```

Chapter 4: Images

Section 4.1: Is "context.drawImage" not displaying the image on the Canvas?

Make sure your image object is fully loaded before you try to draw it on the canvas with `context.drawImage`. Otherwise the image will silently fail to display.

In JavaScript, images are not loaded immediately. Instead, images are loaded asynchronously and during the time they take to load JavaScript continues executing any code that follows `image.src`. This means `context.drawImage` may be executed with an empty image and therefore will display nothing.

Example making sure the image is fully loaded before trying to draw it with .drawImage

```
var img=new Image();
img.onload=start;
img.onerror=function(){alert(img.src+' failed');}
img.src="someImage.png";
function start(){
    // start() is called AFTER the image is fully loaded regardless
    //     of start's position in the code
}
```

Example loading multiple images before trying to draw with any of them

There are more full-functioned image loaders, but this example illustrates how to do it

```
// first image
var img1=new Image();
img1.onload=start;
img1.onerror=function(){alert(img1.src+' failed to load.');};
img1.src="imageOne.png";
// second image
var img2=new Image();
img2.onload=start;
img1.onerror=function(){alert(img2.src+' failed to load.');};
img2.src="imageTwo.png";
//
var imgCount=2;
// start is called every time an image loads
function start(){
    // countdown until all images are loaded
    if(--imgCount>0){return;}
    // All the images are now successfully loaded
    // context.drawImage will successfully draw each one
    context.drawImage(img1,0,0);
    context.drawImage(img2,50,0);
}
```

Section 4.2: The Tained canvas

When adding content from sources outside your domain, or from the local file system the canvas is marked as tainted. Attempt to access the pixel data, or convert to a dataURL will throw a security error.

```
vr image = new Image();
image.src = "file://myLocalImage.png";
```

```
image.onload = function(){
    ctx.drawImage(this,0,0);
    ctx.getImageData(0,0,canvas.width,canvas.height);  // throws a security error
}
```

This example is just a stub to entice someone with a detailed understanding elaborate.

Section 4.3: Image cropping using canvas

This example shows a simple image cropping function that takes an image and cropping coordinates and returns the cropped image.

```
function cropImage(image, croppingCoords) {
    var cc = croppingCoords;
    var workCan = document.createElement("canvas"); // create a canvas
    workCan.width = Math.floor(cc.width);  // set the canvas resolution to the cropped image size
    workCan.height = Math.floor(cc.height);
    var ctx = workCan.getContext("2d");    // get a 2D rendering interface
    ctx.drawImage(image, -Math.floor(cc.x), -Math.floor(cc.y)); // draw the image offset to place
it correctly on the cropped region
    image.src = workCan.toDataURL();       // set the image source to the canvas as a data URL
    return image;
}
```

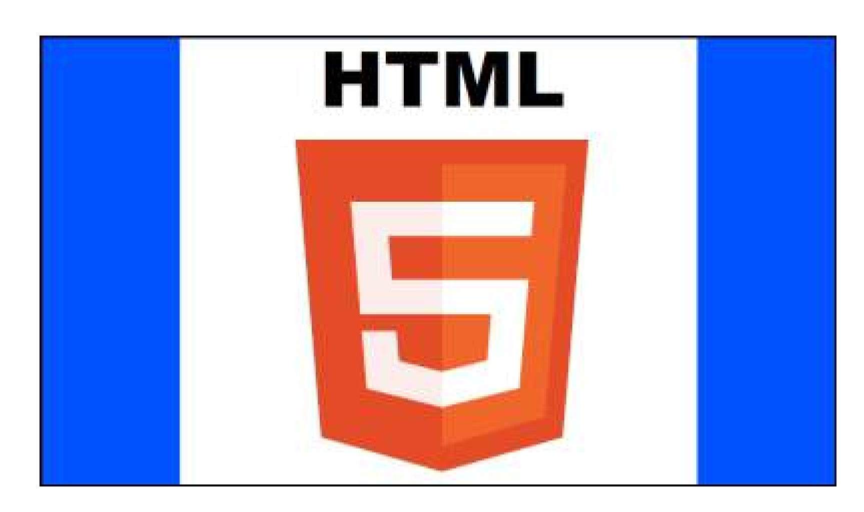
To use

```
var image = new Image();
image.src = "image URL"; // load the image
image.onload = function () {  // when loaded
    cropImage(
        this, {
        x : this.width / 4,     // crop keeping the center
        y : this.height / 4,
        width : this.width / 2,
        height : this.height / 2,
    });
    document.body.appendChild(this);  // Add the image to the DOM
};
```

Section 4.4: Scaling image to fit or fill

Scaling to fit

Means that the whole image will be visible but there may be some empty space on the sides or top and bottom if the image is not the same aspect as the canvas. The example shows the image scaled to fit. The blue on the sides is due to the fact that the image is not the same aspect as the canvas.



Scaling to fill

Means that the image is scaled so that all the canvas pixels will be covered by the image. If the image aspect is not the same as the canvas then some parts of the image will be clipped. The example shows the image scaled to fill. Note how the top and bottom of the image are no longer visible.



Example Scale to fit

```
var image = new Image();
image.src = "imgURL";
image.onload = function(){
    scaleToFit(this);
}

function scaleToFit(img){
    // get the scale
    var scale = Math.min(canvas.width / img.width, canvas.height / img.height);
    // get the top left position of the image
    var x = (canvas.width / 2) - (img.width / 2) * scale;
    var y = (canvas.height / 2) - (img.height / 2) * scale;
    ctx.drawImage(img, x, y, img.width * scale, img.height * scale);
}
```

Example Scale to fill

```
var image = new Image();
image.src = "imgURL";
image.onload = function(){
    scaleToFill(this);
}

function scaleToFill(img){
```

```
    // get the scale
    var scale = Math.max(canvas.width / img.width, canvas.height / img.height);
    // get the top left position of the image
    var x = (canvas.width / 2) - (img.width / 2) * scale;
    var y = (canvas.height / 2) - (img.height / 2) * scale;
    ctx.drawImage(img, x, y, img.width * scale, img.height * scale);
}
```

The only differance between the two functions is getting the scale. The fit uses the min fitting scale will the fill uses the max fitting scale.

Chapter 5: Path (Syntax only)

Section 5.1: createPattern (creates a path styling object)

```
var pattern = createPattern(imageObject,repeat)
```

Creates a reusable pattern (object).

The object can be assigned to any `strokeStyle` and/or `fillStyle`.

Then stroke() or fill() will paint the Path with the pattern of the object.

Arguments:

- **imageObject** is an image that will be used as a pattern. The source of the image can be:
  - HTMLImageElement --- a img element or a new Image(),
  - HTMLCanvasElement --- a canvas element,
  - HTMLVideoElement --- a video element (will grab the current video frame)
  - ImageBitmap,
  - Blob.
- **repeat** determines how the imageObject will be repeated across the canvas (much like a CSS background). This argument must be quote delimited and valid values are:
  - "repeat" --- the pattern will horizontally & vertically fill the canvas
  - "repeat-x" --- the pattern will only repeat horizontally (1 horizontal row)
  - "repeat-y" --- the pattern will only repeat vertically (1 vertical row)
  - "repeat none" --- the pattern appears only once (on the top left)

The pattern object is an object that you can use (and reuse!) to make your path strokes and fills become patterned.

Side Note: The pattern object is not internal to the Canvas element nor it's Context. It is a separate and reusable JavaScript object that you can assign to any Path you desire. You can even use this object to apply pattern to a Path on a different Canvas element(!)

Important hint about Canvas patterns!

When you create a pattern object, the entire canvas is "invisibly" filled with that pattern (subject to the `repeat` argument).

When you `stroke()` or `fill()` a path, the invisible pattern is revealed, but only revealed over that path being stroked or filled.

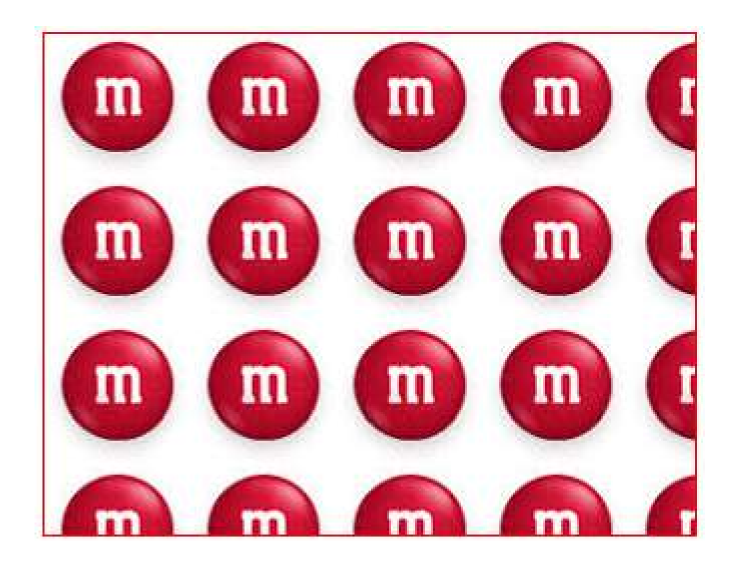
1. Start with an image that you want to use as a pattern. Important(!): Be sure your image has fully loaded (using `patternimage.onload`) before you attempt to use it to create your pattern.



2. You create a pattern like this:

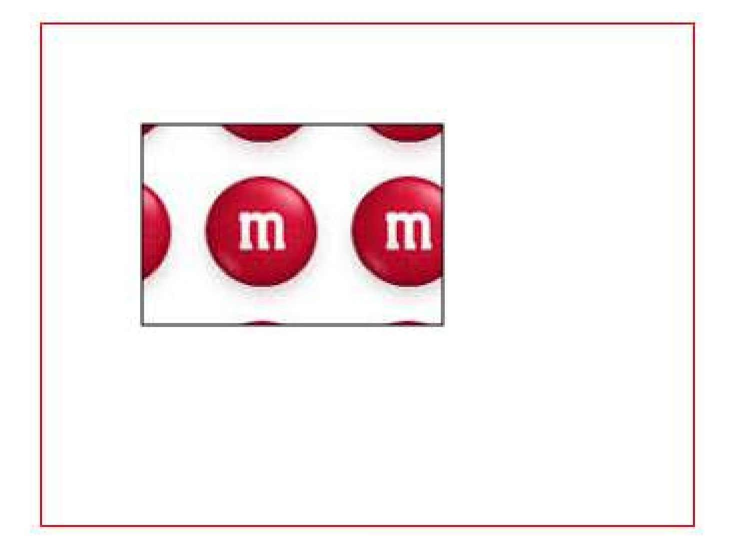
```
// create a pattern
var pattern = ctx.createPattern(patternImage,'repeat');
ctx.fillStyle=pattern;
```

3. Then Canvas will "invisibly" see your pattern creation like this:



4. But until you `stroke()` or `fill()` with the pattern, you will see none of the pattern on the Canvas.

5. Finally, if you stroke or fill a path using the pattern, the "invisible" pattern becomes visible on the Canvas ... but only where the path is drawn.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
```

```
    // fill using a pattern
    var patternImage=new Image();
    // IMPORTANT!
    // Always use .onload to be sure the image has
    //     fully loaded before using it in .createPattern
    patternImage.onload=function(){
        // create a pattern object
        var pattern = ctx.createPattern(patternImage,'repeat');
        // set the fillstyle to that pattern
        ctx.fillStyle=pattern;
        // fill a rectangle with the pattern
        ctx.fillRect(50,50,150,100);
        // demo only, stroke the rect for clarity
        ctx.strokeRect(50,50,150,100);
    }
    patternImage.src='http://i.stack.imgur.com/K9EZl.png';

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=325 height=250></canvas>
</body>
</html>
```

Section 5.2: stroke (a path command)

```
context.stroke()
```

Causes the perimeter of the Path to be stroked according to the current `context.strokeStyle` and the stroked Path is visually drawn onto the canvas.

Prior to executing `context.stroke` (or `context.fill`) the Path exists in memory and is not yet visually drawn on the canvas.

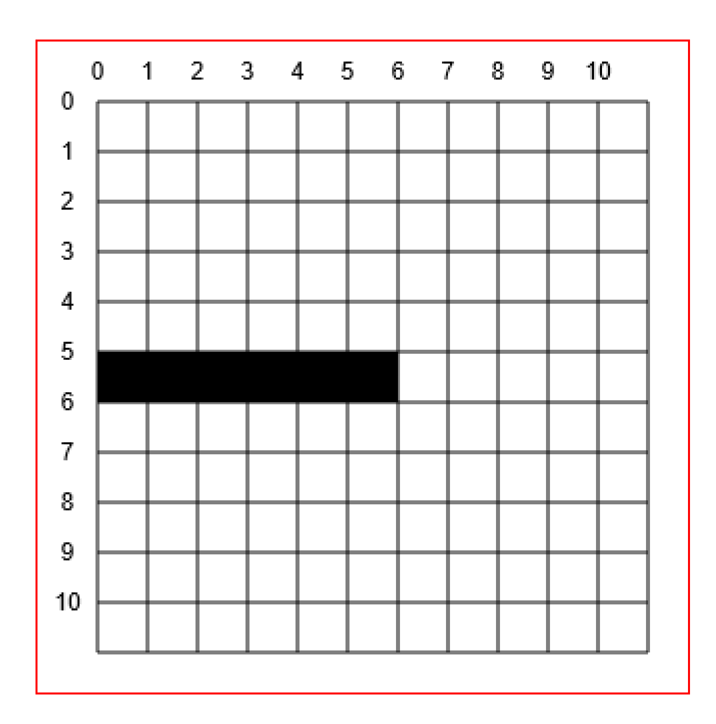
The unusual way strokes are drawn

Consider this example Path that draws a 1 pixel black line from `[0,5]` to `[5,5]`:

```
// draw a 1 pixel black line from [0,5] to [5,5]
context.strokeStyle='black';
context.lineWidth=1;
context.beginPath();
context.moveTo(0,5);
context.lineTo(5,5);
context.stroke();
```

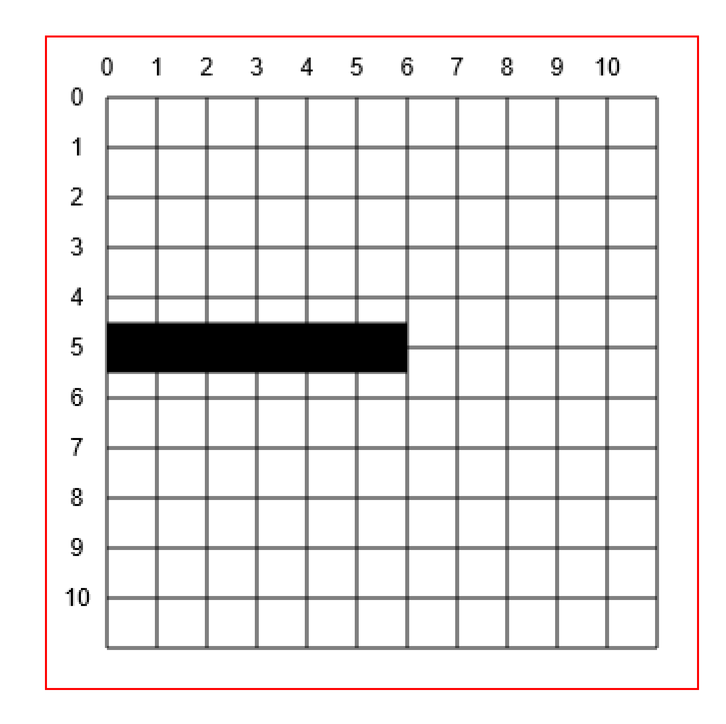
Question: What does the browser actually draw on the canvas?

You probably expect to get 6 black pixels on y=5



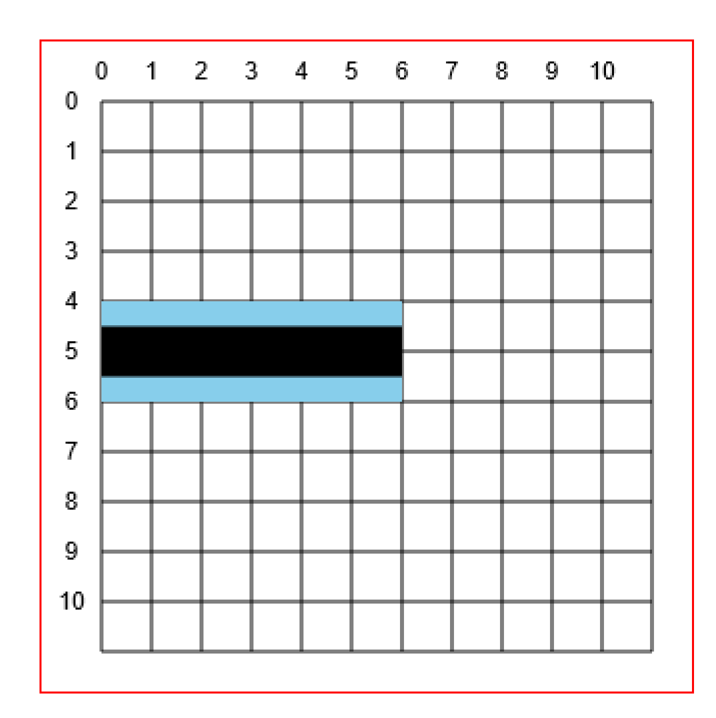
But(!) ... Canvas always draws strokes half-way to either side of the it's defined path!

So since the line is defined at `y==5.0` Canvas wants to draw the line between `y==4.5` and `y==5.5`

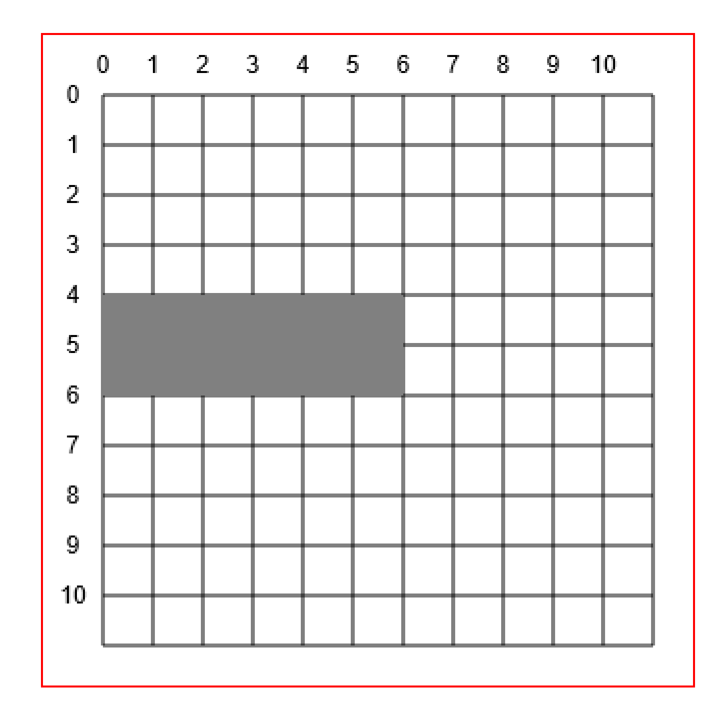


But, again(!) ... The computer display cannot draw half-pixels!

So what is to be done with the undesired half-pixels (shown in blue below)?



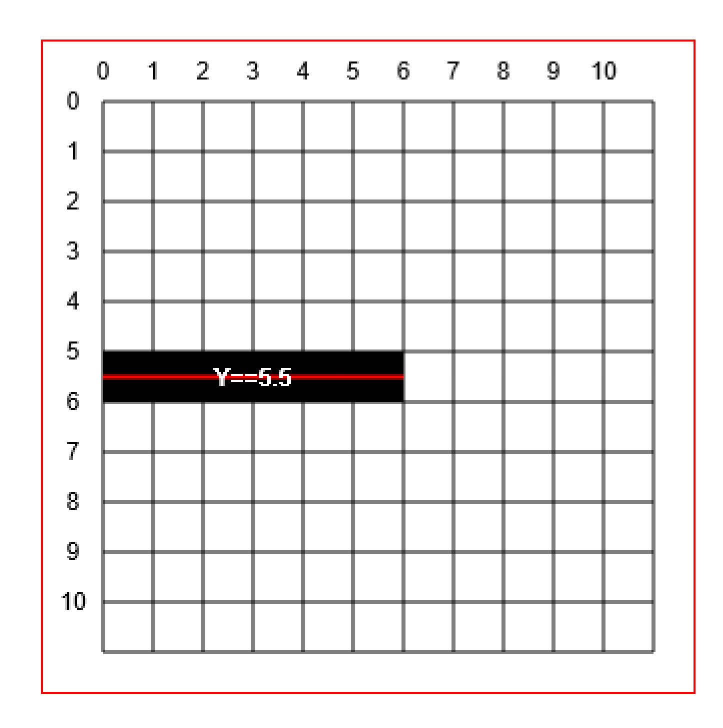
The answer is that Canvas actually orders the display to draw a 2 pixel wide line from `4.0` to `6.0`. It also colors the line lighter than the defined `black`. This strange drawing behavior is "anti-aliasing" and it helps Canvas avoid drawing strokes that look jagged.



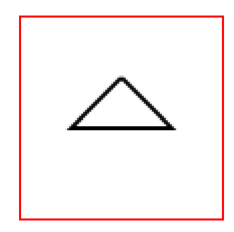
An adjusting trick that ONLY works for exactly horizontal and vertical strokes

You can get a 1 pixel solid black line by specifying the line be drawn on the half-pixel:

```
context.moveTo(0,5.5);
context.lineto(5,5.5);
```



Example code using `context.stroke()` to draw a stroked Path on the canvas:



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    ctx.beginPath();
    ctx.moveTo(50,30);
    ctx.lineTo(75,55);
    ctx.lineTo(25,55);
    ctx.lineTo(50,30);
    ctx.lineWidth=2;
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=100 height=100></canvas>
</body>
</html>
```

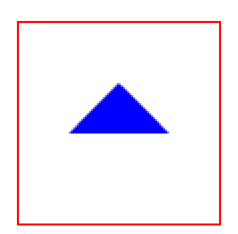
Section 5.3: fill (a path command)

```
context.fill()
```

Causes the inside of the Path to be filled according to the current `context.fillStyle` and the filled Path is visually drawn onto the canvas.

Prior to executing `context.fill` (or `context.stroke`) the Path exists in memory and is not yet visually drawn on the canvas.

Example code using `context.fill()` to draw a filled Path on the canvas:



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    ctx.beginPath();
    ctx.moveTo(50,30);
    ctx.lineTo(75,55);
    ctx.lineTo(25,55);
    ctx.lineTo(50,30);
    ctx.fillStyle='blue';
    ctx.fill();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=100 height=100></canvas>
</body>
</html>
```

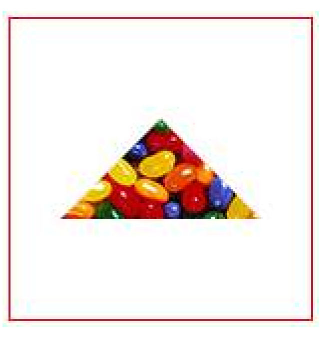
Section 5.4: clip (a path command)

```
context.clip
```

Limits any future drawings to display only inside the current Path.

Example: Clip this image into a triangular Path





```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    var img=new Image();
    img.onload=start;
    img.src='http://i.stack.imgur.com/1CqWf.jpg'

    function start(){
        // draw a triangle path
        ctx.beginPath();
        ctx.moveTo(75,50);
        ctx.lineTo(125,100);
        ctx.lineTo(25,100);
        ctx.lineTo(75,50);

        // clip future drawings to appear only in the triangle
        ctx.clip();

        // draw an image
        ctx.drawImage(img,0,0);
    }

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=150 height=150></canvas>
</body>
</html>
```

Section 5.5: Overview of the basic path drawing commands: lines and curves

Path

A path defines a set of lines and curves which can be visibly drawn on the Canvas.

A path is not automatically drawn on the Canvas. But the path's lines & curves can be drawn onto the Canvas using a styleable stroke. And the shape created by the lines and curves can also be filled with a styleable fill.

Paths have uses beyond drawing on the Canvas:

- Hit testing if an x,y coordinate is inside the path shape.
- Defining a clipping region where only drawings inside the clipping region will be visible. Any drawings outside the clipping region will not be drawn (==transparent) -- similar to CSS overflow.

The basic path drawing commands are:

- beginPath
- moveTo
- lineTo
- arc
- quadraticCurveTo
- bezierCurveTo
- arcTo
- rect
- closePath

Description of the basic drawing commands:

beginPath

```
context.beginPath()
```

Begins assembling a new set of path commands and also discards any previously assembled path.

The discarding is an important and often overlooked point. If you don't begin a new path, any previously issued path commands will automatically be redrawn.

It also moves the drawing "pen" to the top-left origin of the canvas (==coordinate[0,0]).

moveTo

```
context.moveTo(startX, startY)
```

Moves the current pen location to the coordinate [startX,startY].

By default all path drawings are connected together. So the ending point of one line or curve is the starting point of the next line or curve. This can cause an unexpected line to be drawn connecting two adjacent drawings. The `context.moveTo` command basically "picks up the drawing pen" and places it at a new coordinate so the automatic connecting line is not drawn.

lineTo

```
context.lineTo(endX, endY)
```

Draws a line segment from the current pen location to coordinate [endX,endY]

You can assemble multiple `.lineTo` commands to draw a polyline. For example, you could assemble 3 line segments to form a triangle.

arc

```
context.arc(centerX, centerY, radius, startingRadianAngle, endingRadianAngle)
```

Draws a circular arc given a centerpoint, radius and starting & ending angles. The angles are expressed as radians. To convert degrees to radians you can use this formula: `radians = degrees * Math.PI / 180;`.

Angle 0 faces directly rightward from the center of the arc. To draw a complete circle you can make endingAngle = startingAngle + 360 degrees (360 degrees == Math.PI*2): `context.arc(10,10,20,0,Math.PI*2);

By default, the arc is drawn clockwise, An optional [true|false] parameter instructs the arc to be drawn counter-clockwise: `context.arc(10,10,20,0,Math.PI*2,true)`

quadraticCurveTo

```
context.quadraticCurveTo(controlX, controlY, endingX, endingY)
```

Draws a quadratic curve starting at the current pen location to a given ending coordinate. Another given control coordinate determines the shape (curviness) of the curve.

bezierCurveTo

```
context.bezierCurveTo(control1X, control1Y, control2X, control2Y, endingX, endingY)
```

Draws a cubic Bezier curve starting at the current pen location to a given ending coordinate. Another 2 given control coordinates determine the shape (curviness) of the curve.

arcTo

```
context.arcTo(pointX1, pointY1, pointX2, pointY2, radius);
```

Draws a circular arc with a given radius. The arc is drawn clockwise inside the wedge formed by the current pen location and given two points: Point1 & Point2.

A line connecting the current pen location and the start of the arc is automatically drawn preceding the arc.

rect

```
context.rect(leftX, topY, width, height)
```

Draws a rectangle given a top-left corner and a width & height.

The `context.rect` is a unique drawing command because it adds disconnected rectangles. These disconnected rectangles are not automatically connected by lines.

closePath

```
context.closePath()
```

Draws a line from the current pen location back to the beginning path coordinate.

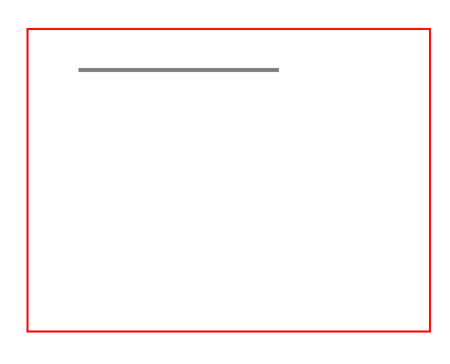
For example, if you draw 2 lines forming 2 legs of a triangle, `closePath` will "close" the triangle by drawing the third leg of the triangle from the 2nd leg's endpoint back to the first leg's starting point.

This command's name often causes it to be misunderstood. `context.closePath` is NOT an ending delimiter to `context.beginPath`. Again, the `closePath` command draws a line -- it does not "close" a `beginPath`.

Section 5.6: lineTo (a path command)

```
context.lineTo(endX, endY)
```

Draws a line segment from the current pen location to coordinate [endX,endY]



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

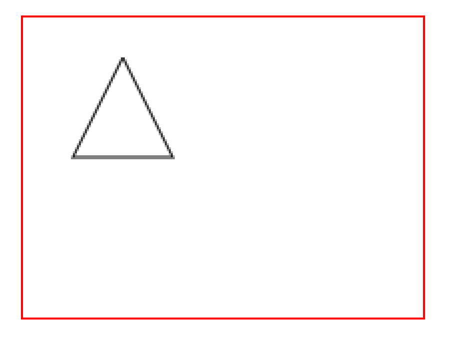
    // arguments
    var startX=25;
    var startY=20;
    var endX=125;
    var endY=20;

    // Draw a single line segment drawn using "moveTo" and "lineTo" commands
    ctx.beginPath();
    ctx.moveTo(startX,startY);
    ctx.lineTo(endX,endY);
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
```

```
</html>
```

You can assemble multiple .lineTo commands to draw a polyline. For example, you could assemble 3 line segments to form a triangle.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // arguments
    var topVertexX=50;
    var topVertexY=20;
    var rightVertexX=75;
    var rightVertexY=70;
    var leftVertexX=25;
    var leftVertexY=70;

    // A set of line segments drawn to form a triangle using
    //     "moveTo" and multiple "lineTo" commands
    ctx.beginPath();
    ctx.moveTo(topVertexX,topVertexY);
    ctx.lineTo(rightVertexX,rightVertexY);
    ctx.lineTo(leftVertexX,leftVertexY);
    ctx.lineTo(topVertexX,topVertexY);
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

Section 5.7: arc (a path command)

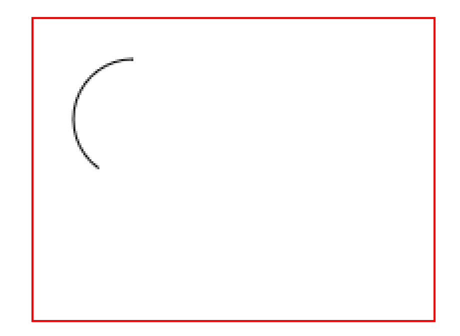
```
context.arc(centerX, centerY, radius, startingRadianAngle, endingRadianAngle)
```

Draws a circular arc given a centerpoint, radius and starting & ending angles. The angles are expressed as radians.

To convert degrees to radians you can use this formula: `radians = degrees * Math.PI / 180;`.

Angle 0 faces directly rightward from the center of the arc.

By default, the arc is drawn clockwise, An optional [true|false] parameter instructs the arc to be drawn counter-clockwise: `context.arc(10,10,20,0,Math.PI*2,true)`



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

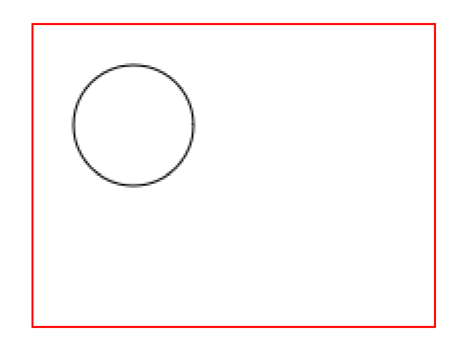
    // get a reference to the canvas element and its context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // arguments
    var centerX=50;
    var centerY=50;
    var radius=30;
    var startingRadianAngle=Math.PI*2*;  // start at 90 degrees == centerY+radius
    var endingRadianAngle=Math.PI*2*.75;  // end at 270 degrees (==PI*2*.75 in radians)

    // A partial circle (i.e. arc) drawn using the "arc" command
    ctx.beginPath();
    ctx.arc(centerX, centerY, radius,  startingRadianAngle, endingRadianAngle);
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

To draw a complete circle you can make endingAngle = startingAngle + 360 degrees (360 degrees == Math.PI2).



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and its context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // arguments
    var centerX=50;
    var centerY=50;
    var radius=30;
    var startingRadianAngle=0;       // start at 0 degrees
    var endingRadianAngle=Math.PI*2; // end at 360 degrees (==PI*2 in radians)

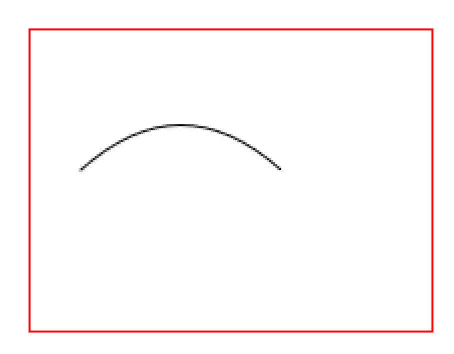
    // A complete circle drawn using the "arc" command
    ctx.beginPath();
    ctx.arc(centerX, centerY, radius,  startingRadianAngle, endingRadianAngle);
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

Section 5.8: quadraticCurveTo (a path command)

```
context.quadraticCurveTo(controlX, controlY, endingX, endingY)
```

Draws a quadratic curve starting at the current pen location to a given ending coordinate. Another given control coordinate determines the shape (curviness) of the curve.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // arguments
    var startX=25;
    var startY=70;
    var controlX=75;
    var controlY=25;
    var endX=125;
    var endY=70;

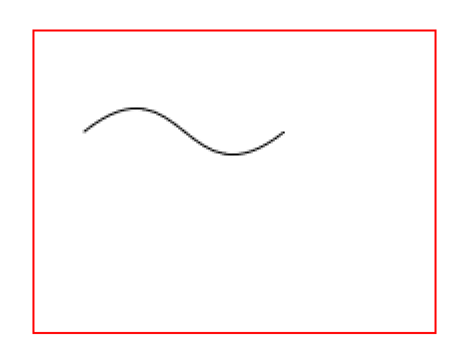
    // A quadratic curve drawn using "moveTo" and "quadraticCurveTo" commands
    ctx.beginPath();
    ctx.moveTo(startX,startY);
    ctx.quadraticCurveTo(controlX,controlY,endX,endY);
    ctx.stroke();


}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

Section 5.9: bezierCurveTo (a path command)

```
context.bezierCurveTo(control1X, control1Y, control2X, control2Y, endingX, endingY)
```

Draws a cubic Bezier curve starting at the current pen location to a given ending coordinate. Another 2 given control coordinates determine the shape (curviness) of the curve.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // arguments
    var startX=25;
    var startY=50;
    var controlX1=75;
    var controlY1=10;
    var controlX2=75;
    var controlY2=90;
    var endX=125;
    var endY=50;

    // A cubic bezier curve drawn using "moveTo" and "bezierCurveTo" commands
    ctx.beginPath();
    ctx.moveTo(startX,startY);
    ctx.bezierCurveTo(controlX1,controlY1,controlX2,controlY2,endX,endY);
    ctx.stroke();

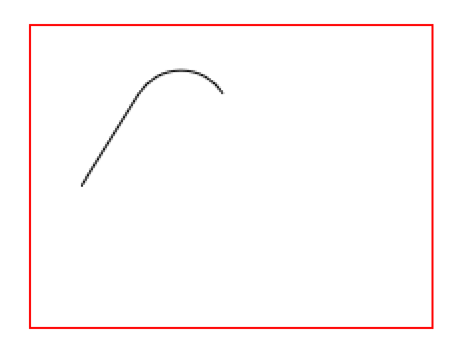
}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

Section 5.10: arcTo (a path command)

```
context.arcTo(pointX1, pointY1, pointX2, pointY2, radius);
```

Draws a circular arc with a given radius. The arc is drawn clockwise inside the wedge formed by the current pen location and given two points: Point1 & Point2.

A line connecting the current pen location and the start of the arc is automatically drawn preceding the arc.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // arguments
    var pointX0=25;
    var pointY0=80;
    var pointX1=75;
    var pointY1=0;
    var pointX2=125;
    var pointY2=80;
    var radius=25;

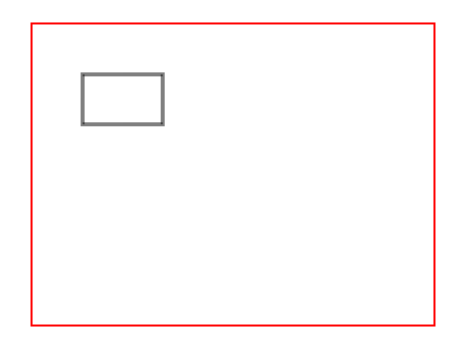
    // A circular arc drawn using the "arcTo" command. The line is automatically drawn.
    ctx.beginPath();
    ctx.moveTo(pointX0,pointY0);
    ctx.arcTo(pointX1, pointY1, pointX2, pointY2, radius);
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

Section 5.11: rect (a path command)

```
context.rect(leftX, topY, width, height)
```

Draws a rectangle given a top-left corner and a width & height.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

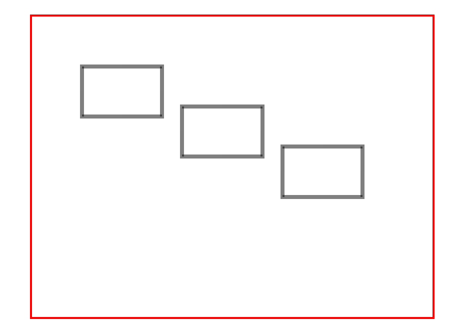
    // arguments
    var leftX=25;
    var topY=25;
    var width=40;
    var height=25;

    // A rectangle drawn using the "rect" command.
    ctx.beginPath();
    ctx.rect(leftX, topY, width, height);
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

The `context.rect` is a unique drawing command because it adds disconnected rectangles.

These disconnected rectangles are not automatically connected by lines.



```
<!doctype html>
<html>
<head>
```

```
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // arguments
    var leftX=25;
    var topY=25;
    var width=40;
    var height=25;

    // Multiple rectangles drawn using the "rect" command.
    ctx.beginPath();
    ctx.rect(leftX, topY, width, height);
    ctx.rect(leftX+50, topY+20, width, height);
    ctx.rect(leftX+100, topY+40, width, height);
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

Section 5.12: closePath (a path command)

```
context.closePath()
```

Draws a line from the current pen location back to the beginning path coordinate.

For example, if you draw 2 lines forming 2 legs of a triangle, closePath will "close" the triangle by drawing the third leg of the triangle from the 2nd leg's endpoint back to the first leg's starting point.

A Misconception explained!

This command's name often causes it to be misunderstood.

`context.closePath` is NOT an ending delimiter to `context.beginPath`.

Again, the closePath command draws a line -- it does not "close" a beginPath.

This example draws 2 legs of a triangle and uses `closePath` to complete (close?!) the triangle by drawing the third leg. What `closePath` is actually doing is drawing a line from the second leg's endpoint back to the first leg's starting point.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // arguments
    var topVertexX=50;
    var topVertexY=50;
    var rightVertexX=75;
    var rightVertexY=75;
    var leftVertexX=25;
    var leftVertexY=75;

    // A set of line segments drawn to form a triangle using
    //     "moveTo" and multiple "lineTo" commands
    ctx.beginPath();
    ctx.moveTo(topVertexX,topVertexY);
    ctx.lineTo(rightVertexX,rightVertexY);
    ctx.lineTo(leftVertexX,leftVertexY);

    // closePath draws the 3rd leg of the triangle
    ctx.closePath()

    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

Section 5.13: beginPath (a path command)

```
context.beginPath()
```

Begins assembling a new set of path commands and also discards any previously assembled path.

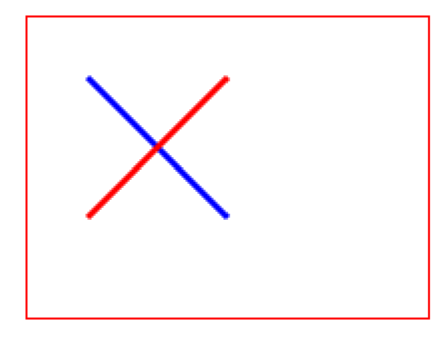
It also moves the drawing "pen" to the top-left origin of the canvas (==coordinate[0,0]).

Although optional, you should ALWAYS start a path with `beginPath`

The discarding is an important and often overlooked point. If you don't begin a new path with `beginPath`, any previously issued path commands will automatically be redrawn.

These 2 demos both attempt to draw an "X" with one red stroke and one blue stroke.

This first demo correctly uses `beginPath` to start it's second red stroke. The result is that the "X" correctly has both a red and a blue stroke.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // draw a blue line
    ctx.beginPath();
    ctx.moveTo(30,30);
    ctx.lineTo(100,100);
    ctx.strokeStyle='blue';
    ctx.lineWidth=3;
    ctx.stroke();

    // draw a red line
    ctx.beginPath();        // Important to begin a new path!
    ctx.moveTo(100,30);
    ctx.lineTo(30,100);
    ctx.strokeStyle='red';
    ctx.lineWidth=3;
    ctx.stroke();

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

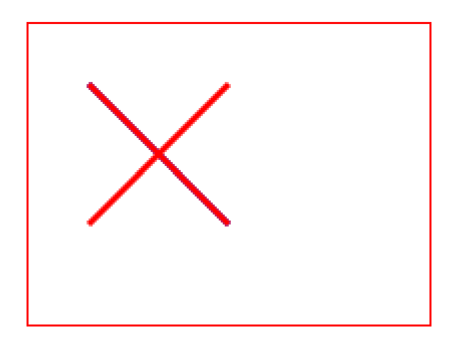
This second demo incorrectly leaves out `beginPath` on the second stroke. The result is that the "X" incorrectly has

both red strokes.

The second `stroke()` is draws the second red stroke.

But without a second `beginPath`, that same second `stroke()` also incorrectly **redraws** the first stroke.

Since the second `stroke()` is now styled as red, the first blue stroke is **overwritten** by an incorrectly colored red stroke.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // get a reference to the canvas element and it's context
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // draw a blue line
    ctx.beginPath();
    ctx.moveTo(30,30);
    ctx.lineTo(100,100);
    ctx.strokeStyle='blue';
    ctx.lineWidth=3;
    ctx.stroke();

    // draw a red line
    // Note: The necessary 'beginPath' is missing!
    ctx.moveTo(100,30);
    ctx.lineTo(30,100);
    ctx.strokeStyle='red';
    ctx.lineWidth=3;
    ctx.stroke();

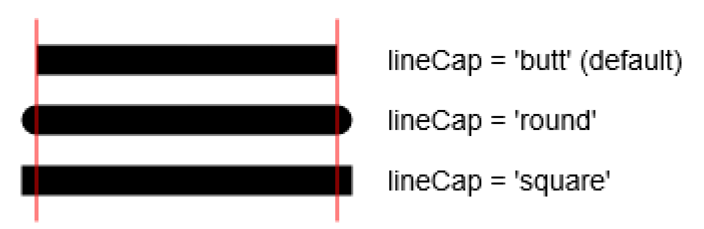
}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=200 height=150></canvas>
</body>
</html>
```

Section 5.14: lineCap (a path styling attribute)

```
context.lineCap=capStyle  // butt (default), round, square
```

Sets the cap style of line starting points and ending points.

- **butt**, the default lineCap style, shows squared caps that do not extend beyond the line's starting and ending points.
- **round**, shows rounded caps that extend beyond the line's starting and ending points.
- **square**, shows squared caps that extend beyond the line's starting and ending points.




```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    ctx.lineWidth=15;

    // lineCap default: butt
    ctx.lineCap='butt';
    drawLine(50,40,200,40);

    // lineCap: round
    ctx.lineCap='round';
    drawLine(50,70,200,70);

    // lineCap: square
    ctx.lineCap='square';
    drawLine(50,100,200,100);

    // utility function to draw a line
    function drawLine(startX,startY,endX,endY){
        ctx.beginPath();
        ctx.moveTo(startX,startY);
        ctx.lineTo(endX,endY);
        ctx.stroke();
    }

    // For demo only,
```

```
    // Rulers to show which lineCaps extend beyond endpoints
    ctx.lineWidth=1;
    ctx.strokeStyle='red';
    drawLine(50,20,50,120);
    drawLine(200,20,200,120);

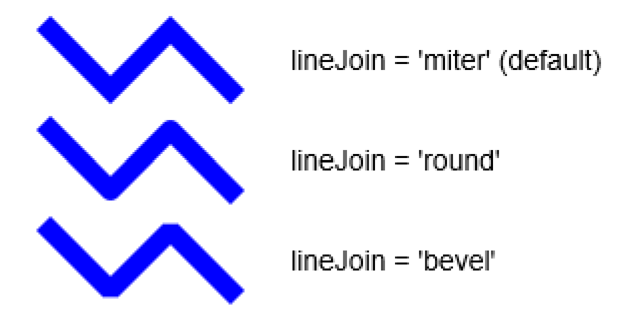
}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=300 height=200></canvas>
</body>
</html>
```

Section 5.15: lineJoin (a path styling attribute)

```
context.lineJoin=joinStyle  // miter (default), round, bevel
```

Sets the style used to connect adjoining line segments.

- **miter**, the default, joins line segments with a sharp joint.
- **round**, joins line segments with a rounded joint.
- **bevel**, joins line segments with a blunted joint.




```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    ctx.lineWidth=15;

    // lineJoin: miter (default)
    ctx.lineJoin='miter';
    drawPolyline(50,30);

    // lineJoin: round
    ctx.lineJoin='round';
    drawPolyline(50,80);

    // lineJoin: bevel
```

```
    ctx.lineJoin='bevel';
    drawPolyline(50,130);

    // utility to draw polyline
    function drawPolyline(x,y){
        ctx.beginPath();
        ctx.moveTo(x,y);
        ctx.lineTo(x+30,y+30);
        ctx.lineTo(x+60,y);
        ctx.lineTo(x+90,y+30);
        ctx.stroke();
    }

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=300 height=200></canvas>
</body>
</html>
```

Section 5.16: strokeStyle (a path styling attribute)

```
context.strokeStyle=color
```

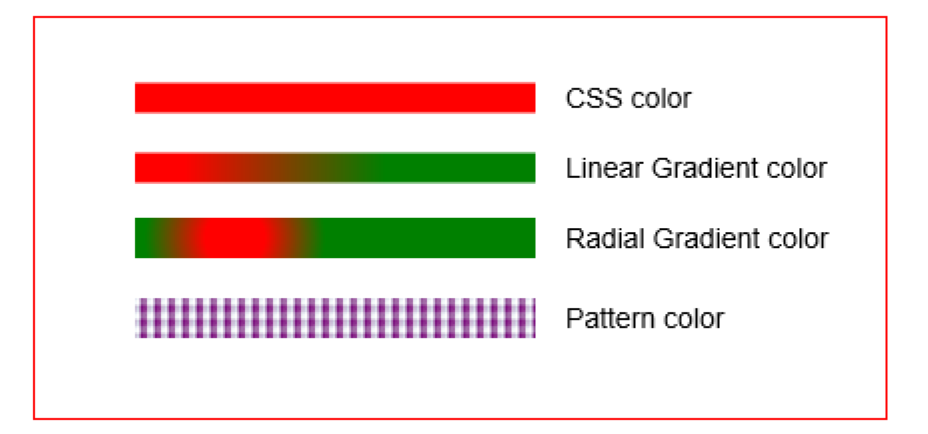
Sets the color that will be used to stroke the outline of the current path.

These are `color` options (these must be quoted):

- **A CSS named color**, for example `context.strokeStyle='red'`
- **A hex color**, for example `context.strokeStyle='#FF0000'`
- **An RGB color**, for example `context.strokeStyle='rgb(red,green,blue)'` where red, green & blue are integers 0-255 indicating the strength of each component color.
- **An HSL color**, for example `context.strokeStyle='hsl(hue,saturation,lightness)'` where hue is an integer 0-360 on the color wheel and saturation & lightness are percentages (0-100%) indicating the strength of each component.
- **An HSLA color**, for example `context.strokeStyle='hsl(hue,saturation,lightness,alpha)'` where hue is an integer 0-360 on the color wheel and saturation & lightness are percentages (0-100%) indicating the strength of each component and alpha is a decimal value 0.00-1.00 indicating the opacity.

You can also specify these color options (these options are objects created by the context):

- **A linear gradient** which is a linear gradient object created with `context.createLinearGradient`
- **A radial gradient** which is a radial gradient object created with `context.createRadialGradient`
- **A pattern** which is a pattern object created with `context.createPattern`




```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    ctx.lineWidth=15;

    // stroke using a CSS color: named, RGB, HSL, etc
    ctx.strokeStyle='red';
    drawLine(50,40,250,40);

    // stroke using a linear gradient
    var gradient = ctx.createLinearGradient(75,75,175,75);
    gradient.addColorStop(0,'red');
    gradient.addColorStop(1,'green');
    ctx.strokeStyle=gradient;
    drawLine(50,75,250,75);

    // stroke using a radial gradient
    var gradient = ctx.createRadialGradient(100,110,15,100,110,45);
    gradient.addColorStop(0,'red');
    gradient.addColorStop(1,'green');
    ctx.strokeStyle=gradient;
    ctx.lineWidth=20;
    drawLine(50,110,250,110);

    // stroke using a pattern
    var patternImage=new Image();
    patternImage.onload=function(){
        var pattern = ctx.createPattern(patternImage,'repeat');
        ctx.strokeStyle=pattern;
        drawLine(50,150,250,150);
    }
    patternImage.src='https://dl.dropboxusercontent.com/u/139992952/stackoverflow/BooMu1.png';

    // for demo only, draw labels by each stroke
    ctx.textBaseline='middle';
    ctx.font='14px arial';
    ctx.fillText('CSS color',265,40);
    ctx.fillText('Linear Gradient color',265,75);
```

```
    ctx.fillText('Radial Gradient color',265,110);
    ctx.fillText('Pattern color',265,150);

    // utility to draw a line
    function drawLine(startX,startY,endX,endY){
        ctx.beginPath();
        ctx.moveTo(startX,startY);
        ctx.lineTo(endX,endY);
        ctx.stroke();
    }

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=425 height=200></canvas>
</body>
</html>
```

Section 5.17: fillStyle (a path styling attribute)

```
context.fillStyle=color
```

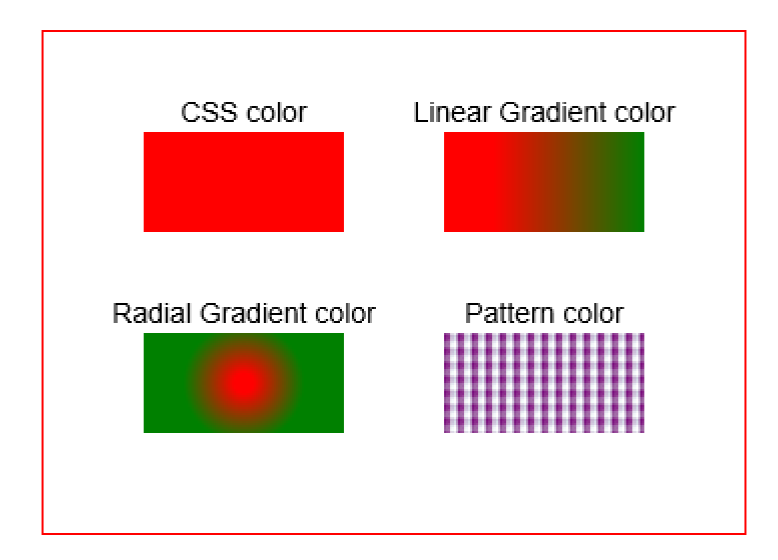
Sets the color that will be used to fill the interior of the current path.

These are color options (these must be quoted):

- **A CSS named color**, for example `context.fillStyle='red'`
- **A hex color**, for example `context.fillStyle='#FF0000'`
- **An RGB color**, for example `context.fillStyle='rgb(red,green,blue)'` where red, green & blue are integers 0-255 indicating the strength of each component color.
- **An HSL color**, for example `context.fillStyle='hsl(hue,saturation,lightness)'` where hue is an integer 0-360 on the color wheel and saturation & lightness are percentages (0-100%) indicating the strength of each component.
- **An HSLA color**, for example `context.fillStyle='hsl(hue,saturation,lightness,alpha)'` where hue is an integer 0-360 on the color wheel and saturation & lightness are percentages (0-100%) indicating the strength of each component and alpha is a decimal value 0.00-1.00 indicating the opacity.

You can also specify these color options (these options are objects created by the context):

- **A linear gradient** which is a linear gradient object created with `context.createLinearGradient`
- **A radial gradient** which is a radial gradient object created with `context.createRadialGradient`
- **A pattern** which is a pattern object created with `context.createPattern`




```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // stroke using a CSS color: named, RGB, HSL, etc
    ctx.fillStyle='red';
    ctx.fillRect(50,50,100,50);

    // stroke using a linear gradient
    var gradient = ctx.createLinearGradient(225,50,300,50);
    gradient.addColorStop(0,'red');
    gradient.addColorStop(1,'green');
    ctx.fillStyle=gradient;
    ctx.fillRect(200,50,100,50);

    // stroke using a radial gradient
    var gradient = ctx.createRadialGradient(100,175,5,100,175,30);
    gradient.addColorStop(0,'red');
    gradient.addColorStop(1,'green');
    ctx.fillStyle=gradient;
    ctx.fillRect(50,150,100,50);

    // stroke using a pattern
    var patternImage=new Image();
    patternImage.onload=function(){
        var pattern = ctx.createPattern(patternImage,'repeat');
        ctx.fillStyle=pattern;
        ctx.fillRect(200,150,100,50);
    }
    patternImage.src='http://i.stack.imgur.com/ixrWe.png';

    // for demo only, draw labels by each stroke
    ctx.fillStyle='black';
    ctx.textAlign='center';
    ctx.textBaseline='middle';
```

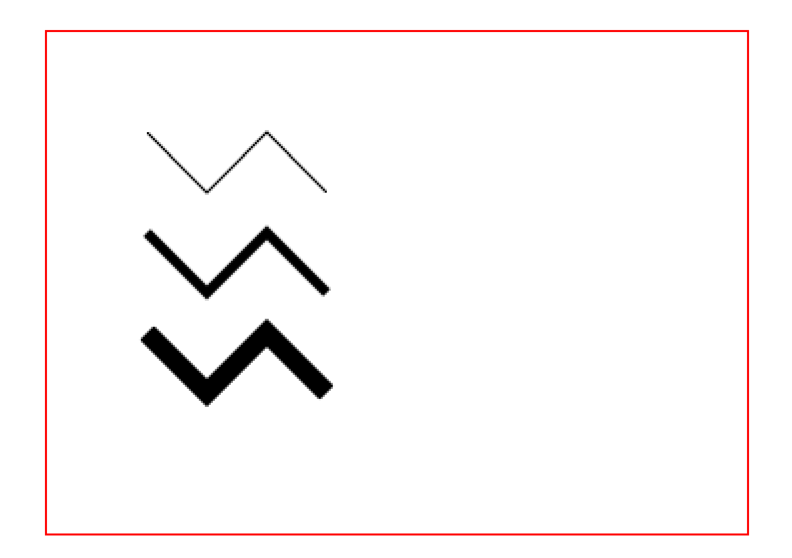
```
    ctx.font='14px arial';
    ctx.fillText('CSS color',100,40);
    ctx.fillText('Linear Gradient color',250,40);
    ctx.fillText('Radial Gradient color',100,140);
    ctx.fillText('Pattern color',250,140);

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=350 height=250></canvas>
</body>
</html>
```

Section 5.18: lineWidth (A path styling attribute)

```
context.lineWidth=lineWidth
```

Sets the width of the line that will stroke the outline of the path



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    ctx.lineWidth=1;
    drawPolyline(50,50);

    ctx.lineWidth=5;
    drawPolyline(50,100);

    ctx.lineWidth=10;
    drawPolyline(50,150);
```

```
    // utility to draw a polyline
    function drawPolyline(x,y){
        ctx.beginPath();
        ctx.moveTo(x,y);
        ctx.lineTo(x+30,y+30);
        ctx.lineTo(x+60,y);
        ctx.lineTo(x+90,y+30);
        ctx.stroke();
    }

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=350 height=250></canvas>
</body>
</html>
```

Section 5.19: shadowColor, shadowBlur, shadowOffsetX, shadowOffsetY (path styling attributes)

```
shadowColor = color        // CSS color
shadowBlur =  width        // integer blur width
shadowOffsetX = distance   // shadow is moved horizontally by this offset
shadowOffsetY = distance   // shadow is moved vertically by this offset
```

This set of attributes will add a shadow around a path.

Both filled paths and stroked paths may have a shadow.

The shadow is darkest (opaque) at the path perimeter and becomes gradiently lighter as it extends away from the path perimeter.

- **shadowColor** indicates which CSS color will be used to create the shadow.
- **shadowBlur** is the distance over which the shadow extends outward from the path.
- **shadowOffsetX** is a distance by which the shadow is shifted horizontally away from the path. A positive distance moves the shadow rightward, a negative distance moves the shadow leftward.
- **shadowOffsetY** is a distance by which the shadow is shifted vertically away from the path. A positive distance moves the shadow downward, a negative distance moves the shadow upward.

About shadowOffsetX & shadowOffsetY

It's important to note that *the whole shadow is shifted in its entirety*. This will cause part of the shadow to shift underneath filled paths and therefore part of the shadow will not be visible.

About shadowed strokes

When shadowing a stroke, both the inside and the outside of the stroke are shadowed. The shadow is darkest at the stroke and lightens as the shadow extends outward in both directions from the stroke.

Turning off shadowing when done

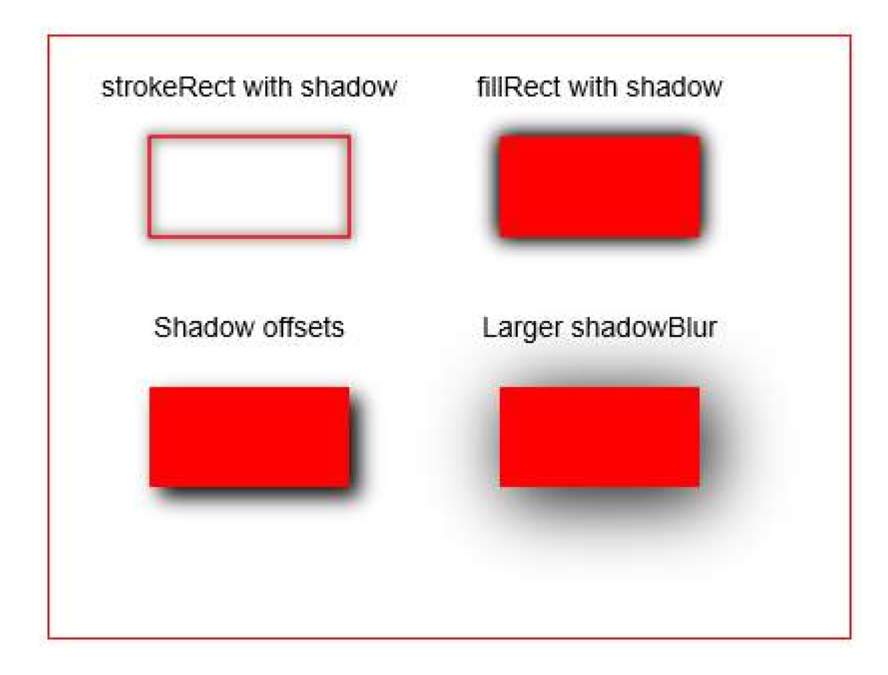
After you have drawn your shadows, you might want to turn shadowing off to draw more paths. To turn shadowing off you set the `shadowColor` to transparent.

```
context.shadowColor = 'rgba(0,0,0,0)';
```

Performance considerations

Shadows (like gradients) requires extensive computations and therefore you should use shadows sparingly.

Be especially cautious when animating because drawing shadows many times per second will greatly impact performance. A workaround if you need to animate shadowed paths is to pre-create the shadowed path on a second "shadow-canvas". The shadow-canvas is a normal canvas that is created in memory with `document.createElement` -- it is not added to the DOM (it's just a staging canvas). Then draw the shadow-canvas onto the main canvas. This is much faster because the shadow computations needn't be made many times per second. All you're doing is copying one prebuilt canvas onto your visible canvas.




```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // shadowed stroke
    ctx.shadowColor='black';
    ctx.shadowBlur=6;
    ctx.strokeStyle='red';
    ctx.strokeRect(50,50,100,50);
    // darken the shadow by stroking a second time
    ctx.strokeRect(50,50,100,50);

    // shadowed fill
    ctx.shadowColor='black';
    ctx.shadowBlur=10;
    ctx.fillStyle='red';
    ctx.fillRect(225,50,100,50);
    // darken the shadow by stroking a second time
    ctx.fillRect(225,50,100,50);
```

```
    // the shadow offset rightward and downward
    ctx.shadowColor='black';
    ctx.shadowBlur=10;
    ctx.shadowOffsetX=5;
    ctx.shadowOffsetY=5;
    ctx.fillStyle='red';
    ctx.fillRect(50,175,100,50);

    // a wider blur (==extends further from the path)
    ctx.shadowColor='black';
    ctx.shadowBlur=35;
    ctx.fillStyle='red';
    ctx.fillRect(225,175,100,50);

    // always clean up! Turn off shadowing
    ctx.shadowColor='rgba(0,0,0,0)';

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=400 height=300></canvas>
</body>
</html>
```

Section 5.20: createLinearGradient (creates a path styling object)

```
var gradient = createLinearGradient( startX, startY, endX, endY )
gradient.addColorStop(gradientPercentPosition, CssColor)
gradient.addColorStop(gradientPercentPosition, CssColor)
[optionally add more color stops to add to the variety of the gradient]
```

Creates a reusable linear gradient (object).

The object can be assigned to any `strokeStyle` and/or `fillStyle`.

Then stroke() or fill() will color the Path with the gradient colors of the object.

Creating a gradient object is a 2-step process:

1. Create the gradient object itself. During creation you define a line on the canvas where the gradient will start and end. The gradient object is created with `var gradient = context.createLinearGradient`.
2. Then add 2 (or more) colors that make up the gradient. This is done by adding multiple color stops to the gradient object with `gradient.addColorStop`.

Arguments:

- **startX,startY** is the canvas coordinate where the gradient starts. At the starting point (and before) the canvas is solidly the color of the lowest `gradientPercentPosition`.
- **endX,endY** is the canvas coordinate where the gradient ends. At the ending point (and after) the canvas is solidly the color of the highest `gradientPercentPosition`.
- **gradientPercentPosition** is a float number between 0.00 and 1.00 assigned to a color stop. It is basically a percentage waypoint along the line where this particular color stop applies.
    - The gradient begins at percentage 0.00 which is [startX,startY] on the canvas.

- The gradient ends at percentage 1.00 which is [endX,endY] on the canvas.
    - *Technical note:* The term "percentage" is not technically correct since the values go from 0.00 to 1.00 rather than 0% to 100%.
- **CssColor** is a CSS color assigned to this particular color stop.

The gradient object is an object that you can use (and reuse!) to make your path strokes and fills become gradient colored.

Side Note: The gradient object is not internal to the Canvas element nor it's Context. It is a separate and reusable JavaScript object that you can assign to any Path you desire. You can even use this object to color a Path on a different Canvas element(!)

Color stops are (percentage) waypoints along the gradient line. At each color stop waypoint, the gradient is fully (==opaquely) colored with it's assigned color. Interim points along the gradient line between color stops are colored as gradients of the this and the previous color.

Important hint about Canvas gradients!

When you create a gradient object, the entire canvas is "invisibly" filled with that gradient.

When you `stroke()` or `fill()` a path, the invisible gradient is revealed, but only revealed over that path being stroked or filled.

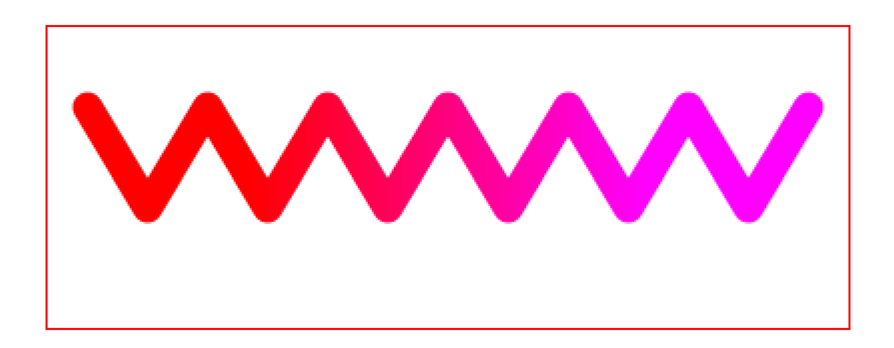
1. If you create a red-to-magenta linear gradient like this:

```
// create a linearGradient
var gradient=ctx.createLinearGradient(100,0,canvas.width-100,0);
gradient.addColorStop(0,'red');
gradient.addColorStop(1,'magenta');
ctx.fillStyle=gradient;
```

2. Then Canvas will "invisibly" see your gradient creation like this:



3. But until you `stroke()` or `fill()` with the gradient, you will see none of the gradient on the Canvas.
4. Finally, if you stroke or fill a path using the gradient, the "invisible" gradient becomes visible on the Canvas ... but only where the path is drawn.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // Create a linearGradient
    // Note: Nothing visually appears during this process
    var gradient=ctx.createLinearGradient(100,0,canvas.width-100,0);
    gradient.addColorStop(0,'red');
    gradient.addColorStop(1,'magenta');

    // Create a polyline path
    // Note: Nothing visually appears during this process
    var x=20;
    var y=40;
    ctx.lineCap='round';
    ctx.lineJoin='round';
    ctx.lineWidth=15;
    ctx.beginPath();
    ctx.moveTo(x,y);
    ctx.lineTo(x+30,y+50);
    ctx.lineTo(x+60,y);
    ctx.lineTo(x+90,y+50);
    ctx.lineTo(x+120,y);
    ctx.lineTo(x+150,y+50);
    ctx.lineTo(x+180,y);
    ctx.lineTo(x+210,y+50);
    ctx.lineTo(x+240,y);
    ctx.lineTo(x+270,y+50);
    ctx.lineTo(x+300,y);
    ctx.lineTo(x+330,y+50);
    ctx.lineTo(x+360,y);

    // Set the stroke style to be the gradient
    // Note: Nothing visually appears during this process
    ctx.strokeStyle=gradient;

    // stroke the path
    // FINALLY! The gradient-stroked path is visible on the canvas
    ctx.stroke();

}); // end window.onload
```

```
</script>
</head>
<body>
    <canvas id="canvas" width=400 height=150></canvas>
</body>
</html>
```

Section 5.21: createRadialGradient (creates a path styling object)

```
var gradient = createRadialGradient(
      centerX1, centerY1, radius1,     // this is the "display' circle
      centerX2, centerY2, radius2      // this is the "light casting" circle
)
gradient.addColorStop(gradientPercentPosition, CssColor)
gradient.addColorStop(gradientPercentPosition, CssColor)
[optionally add more color stops to add to the variety of the gradient]
```

Creates a reusable radial gradient (object). The gradient object is an object that you can use (and reuse!) to make your path strokes and fills become gradient colored.

About...

The Canvas radial gradient is *extremely different* from traditional radial gradients.

The "official" (almost undecipherable!) definition of Canvas's radial gradient is at the bottom of this posting. Don't look at it if you have a weak disposition!!

In (almost understandable) terms:

- The radial gradient has 2 circles: a "casting" circle and a "display" circle.
- The casting circle casts light into the display circle.
- That light is the gradient.
- The shape of that gradient light is determined by the relative size and position of both circles.

Creating a gradient object is a 2-step process:

1. Create the gradient object itself. During creation you define a line on the canvas where the gradient will start and end. The gradient object is created with `var gradient = context.radialLinearGradient`.
2. Then add 2 (or more) colors that make up the gradient. This is done by adding multiple color stops to the gradient object with `gradient.addColorStop`.

Arguments:

- **centerX1,centerY1,radius1** defines a first circle where the gradient will be displayed.
- **centerX2,centerY2,radius2** defines a second circle which is casting gradient light into the first circle.
- **gradientPercentPosition** is a float number between 0.00 and 1.00 assigned to a color stop. It is basically a percentage waypoint defining where this particular color stop applies along the gradient.
    - The gradient begins at percentage 0.00.
    - The gradient ends at percentage 1.00.
    - *Technical note:* The term "percentage" is not technically correct since the values go from 0.00 to 1.00 rather than 0% to 100%.

- **CssColor** is a CSS color assigned to this particular color stop.

Side Note: The gradient object is not internal to the Canvas element nor it's Context. It is a separate and reusable JavaScript object that you can assign to any Path you desire. You can even use this object to color a Path on a different Canvas element(!)

Color stops are (percentage) waypoints along the gradient line. At each color stop waypoint, the gradient is fully (==opaquely) colored with it's assigned color. Interim points along the gradient line between color stops are colored as gradients of the this and the previous color.

Important hint about Canvas gradients!

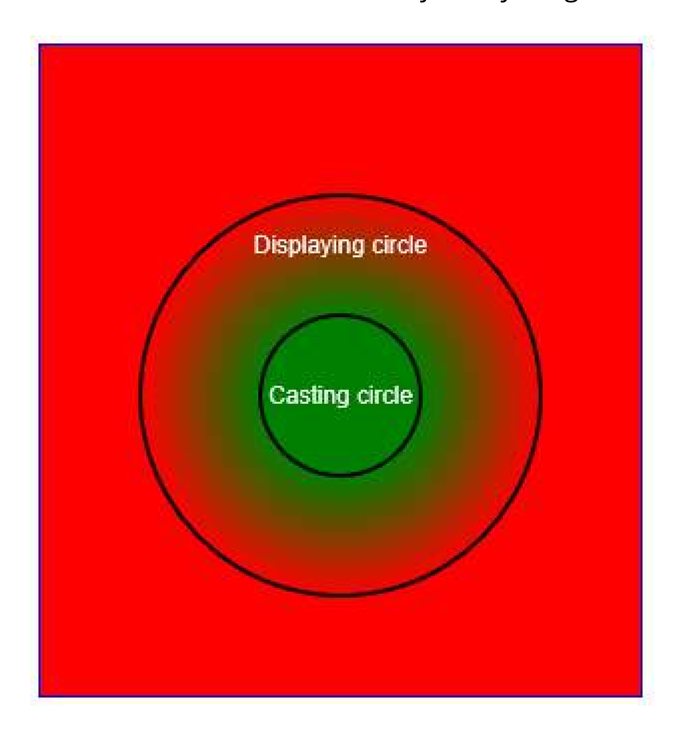
When you create a gradient object, the entire radial gradient is "invisibly" cast upon the canvas.

When you `stroke()` or `fill()` a path, the invisible gradient is revealed, but only revealed over that path being stroked or filled.

1. If you create a green-to-red radial gradient like this:

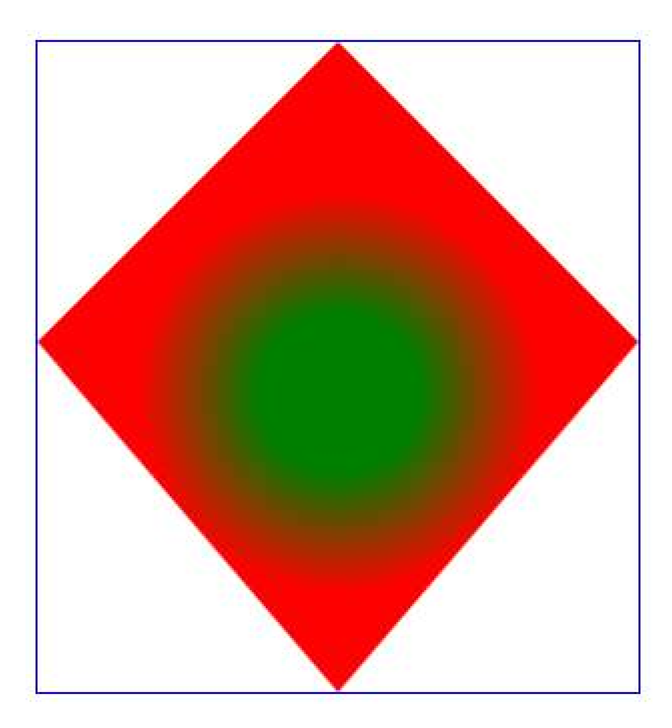
```
// create a radialGradient
var x1=150;
var y1=150;
var x2=280;
var y2=150;
var r1=100;
var r2=120;
var gradient=ctx.createRadialGradient(x1,y1,r1,x2,y2,r2);
gradient.addColorStop(0,'red');
gradient.addColorStop(1,'green');
ctx.fillStyle=gradient;
```

2. Then Canvas will "invisibly" see your gradient creation like this:



3. But until you `stroke()` or `fill()` with the gradient, you will see none of the gradient on the Canvas.
4. Finally, if you stroke or fill a path using the gradient, the "invisible" gradient becomes visible on the Canvas ...

but only where the path is drawn.



```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; padding:10px; }
    #canvas{border:1px solid blue; }
</style>
<script>
window.onload=(function(){

    // canvas related vars
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");

    // create a radial gradient
    var x1=150;
    var y1=175;
    var x2=350;
    var y2=175;
    var r1=100;
    var r2=40;
    x2=x1;
    var gradient=ctx.createRadialGradient(x1,y1,r1,x2,y2,r2);
    gradient.addColorStop(0,'red');
    gradient.addColorStop(1,'green');
    ctx.fillStyle=gradient;

    // fill a path with the gradient
    ctx.beginPath();
    ctx.moveTo(150,0);
    ctx.lineTo(300,150);
    ctx.lineTo(150,325);
    ctx.lineTo(0,150);
    ctx.lineTo(150,0);
    ctx.fill();

}); // end window.onload
</script>
```

```
</head>
<body>
    <canvas id="canvas" width=300 height=325></canvas>
</body>
</html>
```

The scary official details

Who decides what `createRadialGradient does?

The W3C issues the official recommended specifications that browsers use to build the Html5 Canvas element.

The W3C specification for `createRadialGradient` cryptically reads like this:

What does `createRadialGradient` create

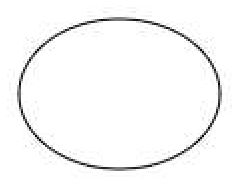
> `createRadialGradient` … effectively creates a cone, touched by the two circles defined in the creation of the gradient, with the part of the cone before the start circle (0.0) using the color of the first offset, the part of the cone after the end circle (1.0) using the color of the last offset, and areas outside the cone untouched by the gradient (transparent black).

How does it work internally

> The `createRadialGradient(x0, y0, r0, x1, y1, r1)` method takes six arguments, the first three representing the start circle with origin (x0, y0) and radius r0, and the last three representing the end circle with origin (x1, y1) and radius r1. The values are in coordinate space units. If either of r0 or r1 are negative, an IndexSizeError exception must be thrown. Otherwise, the method must return a radial CanvasGradient initialized with the two specified circles.
>
> Radial gradients must be rendered by following these steps:
>
> 1. If x0 = x1 and y0 = y1 and r0 = r1, then the radial gradient must paint nothing. Abort these steps.
> 2. Let x(ω) = (x1-x0)ω + x0; Let y(ω) = (y1-y0)ω + y0; Let r(ω) = (r1-r0)ω + r0 Let the color at ω be the color at that position on the gradient (with the colors coming from the interpolation and extrapolation described above).
> 3. For all values of ω where r(ω) > 0, starting with the value of ω nearest to positive infinity and ending with the value of ω nearest to negative infinity, draw the circumference of the circle with radius r(ω) at position (x(ω), y(ω)), with the color at ω, but only painting on the parts of the canvas that have not yet been painted on by earlier circles in this step for this rendering of the gradient.

Chapter 6: Paths

Section 6.1: Ellipse



Note: Browsers are in the process of adding a built-in `context.ellipse` drawing command, but this command is not universally adopted (notably not in IE). The methods below work in all browsers.

Draw an ellipse given it's desired top-left coordinate:

```
// draws an ellipse based on x,y being top-left coordinate
function drawEllipse(x,y,width,height){
    var PI2=Math.PI*2;
    var ratio=height/width;
    var radius=Math.max(width,height)/2;
    var increment = 1 / radius;
    var cx=x+width/2;
    var cy=y+height/2;

    ctx.beginPath();
    var x = cx + radius * Math.cos(0);
    var y = cy - ratio * radius * Math.sin(0);
    ctx.lineTo(x,y);

    for(var radians=increment; radians<PI2; radians+=increment){
        var x = cx + radius * Math.cos(radians);
        var y = cy - ratio * radius * Math.sin(radians);
        ctx.lineTo(x,y);
     }

    ctx.closePath();
    ctx.stroke();
}
```

Draw an ellipse given it's desired center point coordinate:

```
// draws an ellipse based on cx,cy being ellipse's centerpoint coordinate
function drawEllipse2(cx,cy,width,height){
    var PI2=Math.PI*2;
    var ratio=height/width;
    var radius=Math.max(width,height)/2;
    var increment = 1 / radius;

    ctx.beginPath();
    var x = cx + radius * Math.cos(0);
    var y = cy - ratio * radius * Math.sin(0);
    ctx.lineTo(x,y);

    for(var radians=increment; radians<PI2; radians+=increment){
        var x = cx + radius * Math.cos(radians);
        var y = cy - ratio * radius * Math.sin(radians);
        ctx.lineTo(x,y);
     }
```

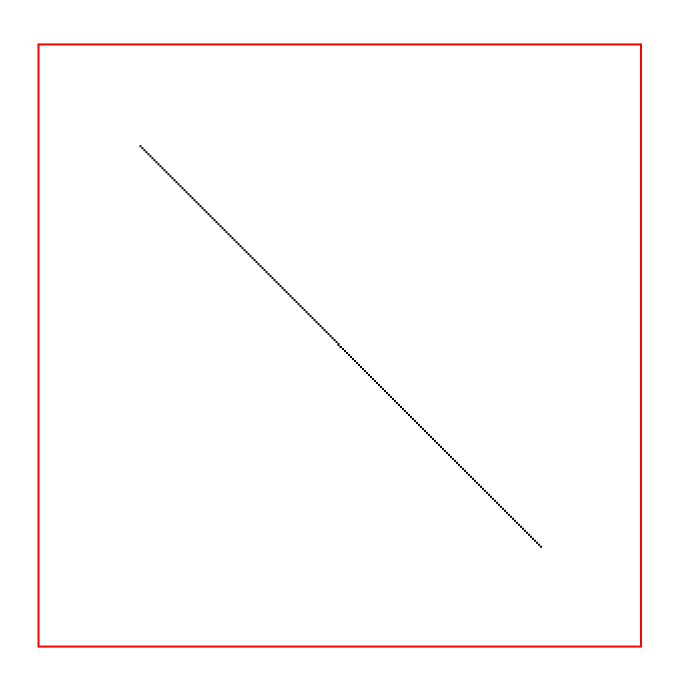
```
    ctx.closePath();
    ctx.stroke();
}
```

Section 6.2: Line without blurryness

When Canvas draws a line it automatically adds anti-aliasing to visually heal "jaggedness". The result is a line that is less jagged but more blurry.

This function draws a line between 2 points without anti-aliasing using Bresenham's_line algorithm. The result is a crisp line without the jaggedness.

Important Note: *This pixel-by-pixel method is a much slower drawing method than* `context.lineTo`.



```
// Usage:
bresenhamLine(50,50,250,250);

// x,y line start
// xx,yy line end
// the pixel at line start and line end are drawn
function bresenhamLine(x, y, xx, yy){
    var oldFill = ctx.fillStyle;  // save old fill style
    ctx.fillStyle = ctx.strokeStyle; // move stroke style to fill
    xx = Math.floor(xx);
    yy = Math.floor(yy);
    x = Math.floor(x);
    y = Math.floor(y);
    // BRENSENHAM
    var dx =  Math.abs(xx-x);
    var sx = x < xx ? 1 : -1;
    var dy = -Math.abs(yy-y);
    var sy = y<yy ? 1 : -1;
    var err = dx+dy;
    var errC; // error value
    var end = false;
    var x1 = x;
    var y1 = y;

    while(!end){
       ctx.fillRect(x1, y1, 1, 1); // draw each pixel as a rect
```

```
        if (x1 === xx && y1 === yy) {
            end = true;
        }else{
            errC = 2*err;
            if (errC >= dy) {
                err += dy;
                x1 += sx;
            }
            if (errC <= dx) {
                err += dx;
                y1 += sy;
            }
        }
    }
    ctx.fillStyle = oldFill; // restore old fill style
}
```

Chapter 7: Navigating along a Path

Section 7.1: Find point on curve

This example finds a point on a bezier or cubic curve at `position` where `position` is he unit distance on the curve 0 <= `position` <= 1. The position is clamped to the range thus if values < 0 or > 1 are passed they will be set 0,1 respectively.

Pass the function 6 coordinates for quadratic bezier or 8 for cubic.

The last optional argument is the returned vector (point). If not given it will be created.

Example usage

```
var p1 = {x : 10 , y : 100};
var p2 = {x : 100, y : 200};
var p3 = {x : 200, y : 0};
var p4 = {x : 300, y : 100};
var point = {x : null, y : null};

// for cubic beziers
point = getPointOnCurve(0.5, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, p4.x, p4.y, point);
// or No need to set point as it is a referance and will be set
getPointOnCurve(0.5, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, p4.x, p4.y, point);
// or to create a new point
var point1 = getPointOnCurve(0.5, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, p4.x, p4.y);

// for quadratic beziers
point = getPointOnCurve(0.5, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, null, null, point);
// or No need to set point as it is a referance and will be set
getPointOnCurve(0.5, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, null, null, point);
// or to create a new point
var point1 = getPointOnCurve(0.5, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y);
```

The function

getPointOnCurve = function(position, x1, y1, x2, y2, x3, y3, [x4, y4], [vec])

> **Note:** Arguments inside [x4, y4] are optional.

> **Note:** `x4`,`y4` if `null`, or `undefined` means that the curve is a quadratic bezier. `vec` is optional and will hold the returned point if supplied. If not it will be created.

```
var getPointOnCurve = function(position, x1, y1, x2, y2, x3, y3, x4, y4, vec){
    var vec, quad;
    quad = false;
    if(vec === undefined){
        vec = {};
    }

    if(x4 === undefined || x4 === null){
        quad = true;
        x4 = x3;
```

```
        y4 = y3;
    }

    if(position <= 0){
        vec.x = x1;
        vec.y = y1;
        return vec;
    }
    if(position >= 1){
        vec.x = x4;
        vec.y = y4;
        return vec;
    }
    c = position;
    if(quad){
        x1 += (x2 - x1) * c;
        y1 += (y2 - y1) * c;
        x2 += (x3 - x2) * c;
        y2 += (y3 - y2) * c;
        vec.x = x1 + (x2 - x1) * c;
        vec.y = y1 + (y2 - y1) * c;
        return vec;
    }
    x1 += (x2 - x1) * c;
    y1 += (y2 - y1) * c;
    x2 += (x3 - x2) * c;
    y2 += (y3 - y2) * c;
    x3 += (x4 - x3) * c;
    y3 += (y4 - y3) * c;
    x1 += (x2 - x1) * c;
    y1 += (y2 - y1) * c;
    x2 += (x3 - x2) * c;
    y2 += (y3 - y2) * c;
    vec.x = x1 + (x2 - x1) * c;
    vec.y = y1 + (y2 - y1) * c;
    return vec;
}
```

Section 7.2: Finding extent of Quadratic Curve

When you need to find the bounding rectangle of a quadratic bezier curve you can use the following performant method.

```
// This method was discovered by Blindman67 and solves by first normalising the control point thereby
reducing the algorithm complexity
// x1,y1, x2,y2, x3,y3 Start, Control, and End coords of bezier
// [extent] is optional and if provided the extent will be added to it allowing you to use the
function
//        to get the extent of many beziers.
// returns extent object (if not supplied a new extent is created)
// Extent object properties
// top, left,right,bottom,width,height
function getQuadraticCurevExtent(x1, y1, x2, y2, x3, y3, extent) {
    var brx, bx, x, bry, by, y, px, py;

    // solve quadratic for bounds by BM67 normalizing equation
    brx = x3 - x1; // get x range
    bx = x2 - x1; // get x control point offset
    x = bx / brx; // normalise control point which is used to check if maxima is in range

    // do the same for the y points
```

```
    bry = y3 - y1;
    by = y2 - y1;
    y = by / bry;

    px = x1; // set defaults in case maximas outside range
    py = y1;

    // find top/left, top/right, bottom/left, or bottom/right
    if (x < 0 || x > 1) { // check if x maxima is on the curve
        px = bx * bx / (2 * bx - brx) + x1; // get the x maxima
    }
    if (y < 0 || y > 1) { // same as x
        py = by * by / (2 * by - bry) + y1;
    }

    // create extent object and add extent
    if (extent === undefined) {
        extent = {};
        extent.left = Math.min(x1, x3, px);
        extent.top = Math.min(y1, y3, py);
        extent.right = Math.max(x1, x3, px);
        extent.bottom = Math.max(y1, y3, py);
    } else { // use spplied extent and extend it to fit this curve
        extent.left = Math.min(x1, x3, px, extent.left);
        extent.top = Math.min(y1, y3, py, extent.top);
        extent.right = Math.max(x1, x3, px, extent.right);
        extent.bottom = Math.max(y1, y3, py, extent.bottom);
    }

    extent.width = extent.right - extent.left;
    extent.height = extent.bottom - extent.top;
    return extent;
}
```

For a more detailed look at solving for extent see answer To get extent of a quadratic bezier which includes runnable demos.

Section 7.3: Finding points along a cubic Bezier curve

This example finds an array of approximately evenly spaced points along a cubic Bezier curve.

It decomposes Path segments created with `context.bezierCurveTo` into points along that curve.

```
// Return: an array of approximately evenly spaced points along a cubic Bezier curve
//
// Attribution: Stackoverflow's @Blindman67
// Cite:
http://stackoverflow.com/questions/36637211/drawing-a-curved-line-in-css-or-canvas-and-moving-circle-
along-it/36827074#36827074
// As modified from the above citation
//
// ptCount: sample this many points at interval along the curve
// pxTolerance: approximate spacing allowed between points
// Ax,Ay,Bx,By,Cx,Cy,Dx,Dy: control points defining the curve
//
function plotCBez(ptCount,pxTolerance,Ax,Ay,Bx,By,Cx,Cy,Dx,Dy){
    var deltaBAx=Bx-Ax;
    var deltaCBx=Cx-Bx;
    var deltaDCx=Dx-Cx;
    var deltaBAy=By-Ay;
```

```
    var deltaCBy=Cy-By;
    var deltaDCy=Dy-Cy;
    var ax,ay,bx,by;
    var lastX=-10000;
    var lastY=-10000;
    var pts=[{x:Ax,y:Ay}];
    for(var i=1;i<ptCount;i++){
        var t=i/ptCount;
        ax=Ax+deltaBAx*t;
        bx=Bx+deltaCBx*t;
        cx=Cx+deltaDCx*t;
        ax+=(bx-ax)*t;
        bx+=(cx-bx)*t;
        //
        ay=Ay+deltaBAy*t;
        by=By+deltaCBy*t;
        cy=Cy+deltaDCy*t;
        ay+=(by-ay)*t;
        by+=(cy-by)*t;
        var x=ax+(bx-ax)*t;
        var y=ay+(by-ay)*t;
        var dx=x-lastX;
        var dy=y-lastY;
        if(dx*dx+dy*dy>pxTolerance){
            pts.push({x:x,y:y});
            lastX=x;
            lastY=y;
        }
    }
    pts.push({x:Dx,y:Dy});
    return(pts);
}
```

Section 7.4: Finding points along a quadratic curve

This example finds an array of approximately evenly spaced points along a quadratic curve.

It decomposes Path segments created with `context.quadraticCurveTo` into points along that curve.

```
// Return: an array of approximately evenly spaced points along a Quadratic curve
//
// Attribution: Stackoverflow's @Blindman67
// Cite:
http://stackoverflow.com/questions/36637211/drawing-a-curved-line-in-css-or-canvas-and-moving-circle-along-it/36827074#36827074
// As modified from the above citation
//
// ptCount: sample this many points at interval along the curve
// pxTolerance: approximate spacing allowed between points
// Ax,Ay,Bx,By,Cx,Cy: control points defining the curve
//
function plotQBez(ptCount,pxTolerance,Ax,Ay,Bx,By,Cx,Cy){
    var deltaBAx=Bx-Ax;
    var deltaCBx=Cx-Bx;
    var deltaBAy=By-Ay;
    var deltaCBy=Cy-By;
    var ax,ay;
    var lastX=-10000;
    var lastY=-10000;
    var pts=[{x:Ax,y:Ay}];
    for(var i=1;i<ptCount;i++){
```

```
        var t=i/ptCount;
        ax=Ax+deltaBAx*t;
        ay=Ay+deltaBAy*t;
        var x=ax+((Bx+deltaCBx*t)-ax)*t;
        var y=ay+((By+deltaCBy*t)-ay)*t;
        var dx=x-lastX;
        var dy=y-lastY;
        if(dx*dx+dy*dy>pxTolerance){
            pts.push({x:x,y:y});
            lastX=x;
            lastY=y;
        }
    }
    pts.push({x:Cx,y:Cy});
    return(pts);
}
```

Section 7.5: Finding points along a line

This example finds an array of approximately evenly spaced points along a line.

It decomposes Path segments created with `context.lineTo` into points along that line.

```
// Return: an array of approximately evenly spaced points along a line
//
// pxTolerance: approximate spacing allowed between points
// Ax,Ay,Bx,By: end points defining the line
//
function plotLine(pxTolerance,Ax,Ay,Bx,By){
    var dx=Bx-Ax;
    var dy=By-Ay;
    var ptCount=parseInt(Math.sqrt(dx*dx+dy*dy))*3;
    var lastX=-10000;
    var lastY=-10000;
    var pts=[{x:Ax,y:Ay}];
    for(var i=1;i<=ptCount;i++){
        var t=i/ptCount;
        var x=Ax+dx*t;
        var y=Ay+dy*t;
        var dx1=x-lastX;
        var dy1=y-lastY;
        if(dx1*dx1+dy1*dy1>pxTolerance){
            pts.push({x:x,y:y});
            lastX=x;
            lastY=y;
        }
    }
    pts.push({x:Bx,y:By});
    return(pts);
}
```

Section 7.6: Finding points along an entire Path containing curves and lines

This example finds an array of approximately evenly spaced points along an entire Path.

It decomposes all Path segments created with `context.lineTo`, `context.quadraticCurveTo` and/or `context.bezierCurveTo` into points along that Path.

Usage

```
// Path related variables
var A={x:50,y:100};
var B={x:125,y:25};
var BB={x:150,y:15};
var BB2={x:150,y:185};
var C={x:175,y:200};
var D={x:300,y:150};
var n=1000;
var tolerance=1.5;
var pts;

// canvas related variables
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
document.body.appendChild(canvas);
canvas.width=378;
canvas.height=256;

// Tell the Context to plot waypoint in addition to
// drawing the path
plotPathCommands(ctx,n,tolerance);

// Path drawing commands
ctx.beginPath();
ctx.moveTo(A.x,A.y);
ctx.bezierCurveTo(B.x,B.y,C.x,C.y,D.x,D.y);
ctx.quadraticCurveTo(BB.x,BB.y,A.x,A.y);
ctx.lineTo(D.x,D.y);
ctx.strokeStyle='gray';
ctx.stroke();

// Tell the Context to stop plotting waypoints
ctx.stopPlottingPathCommands();

// Demo: Incrementally draw the path using the plotted points
ptsToRects(ctx.getPathPoints());
function ptsToRects(pts){
    ctx.fillStyle='red';
    var i=0;
    requestAnimationFrame(animate);
    function animate(){
        ctx.fillRect(pts[i].x-0.50,pts[i].y-0.50,tolerance,tolerance);
        i++;
        if(i<pts.length){ requestAnimationFrame(animate); }
    }
}
```

An plug-in that automatically calculates points along the path

This code modifies these Canvas Context's drawing commands so the commands not only draw the line or curve, but also create an array of points along the entire path:

- beginPath,
- moveTo,
- lineTo,
- quadraticCurveTo,
- bezierCurveTo.

Important Note!

This code modifies the actual drawing functions of the Context so when you are done plotting points along the path, you should call the supplied `stopPlottingPathCommands` to return the Context drawing functions to their unmodified state.

The purpose of this modified Context is to allow you to "plug-in" the points-array calculation into your existing code without having to modify your existing Path drawing commands. But, you don't need to use this modified Context -- you can separately call the individual functions that decompose a line, a quadratic curve and a cubic Bezier curve and then manually concatenate those individual point-arrays into a single point-array for the entire path.

You fetch a copy of the resulting points-array using the supplied `getPathPoints` function.

If you draw multiple Paths with the modified Context, the points-array will contain a single concatenated set of points for all the multiple Paths drawn.

If, instead, you want to get separate points-arrays, you can fetch the current array with `getPathPoints` and then clear those points from the array with the supplied `clearPathPoints` function.

```
// Modify the Canvas' Context to calculate a set of approximately
//     evenly spaced waypoints as it draws path(s).
function plotPathCommands(ctx,sampleCount,pointSpacing){
    ctx.mySampleCount=sampleCount;
    ctx.myPointSpacing=pointSpacing;
    ctx.myTolerance=pointSpacing*pointSpacing;
    ctx.myBeginPath=ctx.beginPath;
    ctx.myMoveTo=ctx.moveTo;
    ctx.myLineTo=ctx.lineTo;
    ctx.myQuadraticCurveTo=ctx.quadraticCurveTo;
    ctx.myBezierCurveTo=ctx.bezierCurveTo;
    // don't use myPathPoints[] directly -- use "ctx.getPathPoints"
    ctx.myPathPoints=[];
    ctx.beginPath=function(){
        this.myLastX=0;
        this.myLastY=0;
        this.myBeginPath();
    }
    ctx.moveTo=function(x,y){
        this.myLastX=x;
        this.myLastY=y;
        this.myMoveTo(x,y);
    }
    ctx.lineTo=function(x,y){
        var pts=plotLine(this.myTolerance,this.myLastX,this.myLastY,x,y);
        Array.prototype.push.apply(this.myPathPoints,pts);
        this.myLastX=x;
        this.myLastY=y;
        this.myLineTo(x,y);
    }
    ctx.quadraticCurveTo=function(x0,y0,x1,y1){
        var
pts=plotQBez(this.mySampleCount,this.myTolerance,this.myLastX,this.myLastY,x0,y0,x1,y1);
        Array.prototype.push.apply(this.myPathPoints,pts);
        this.myLastX=x1;
        this.myLastY=y1;
        this.myQuadraticCurveTo(x0,y0,x1,y1);
    }
    ctx.bezierCurveTo=function(x0,y0,x1,y1,x2,y2){
        var
```

```
pts=plotCBez(this.mySampleCount,this.myTolerance,this.myLastX,this.myLastY,x0,y0,x1,y1,x2,y2);
        Array.prototype.push.apply(this.myPathPoints,pts);
        this.myLastX=x2;
        this.myLastY=y2;
        this.myBezierCurveTo(x0,y0,x1,y1,x2,y2);
    }
    ctx.getPathPoints=function(){
        return(this.myPathPoints.slice());
    }
    ctx.clearPathPoints=function(){
        this.myPathPoints.length=0;
    }
    ctx.stopPlottingPathCommands=function(){
        if(!this.myBeginPath){return;}
        this.beginPath=this.myBeginPath;
        this.moveTo=this.myMoveTo;
        this.lineTo=this.myLineTo;
        this.quadraticCurveto=this.myQuadraticCurveTo;
        this.bezierCurveTo=this.myBezierCurveTo;
        this.myBeginPath=undefined;
    }
}
```

A complete Demo:

```
// Path related variables
var A={x:50,y:100};
var B={x:125,y:25};
var BB={x:150,y:15};
var BB2={x:150,y:185};
var C={x:175,y:200};
var D={x:300,y:150};
var n=1000;
var tolerance=1.5;
var pts;

// canvas related variables
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
document.body.appendChild(canvas);
canvas.width=378;
canvas.height=256;

// Tell the Context to plot waypoint in addition to
// drawing the path
plotPathCommands(ctx,n,tolerance);

// Path drawing commands
ctx.beginPath();
ctx.moveTo(A.x,A.y);
ctx.bezierCurveTo(B.x,B.y,C.x,C.y,D.x,D.y);
ctx.quadraticCurveTo(BB.x,BB.y,A.x,A.y);
ctx.lineTo(D.x,D.y);
ctx.strokeStyle='gray';
ctx.stroke();

// Tell the Context to stop plotting waypoints
ctx.stopPlottingPathCommands();

// Incrementally draw the path using the plotted points
ptsToRects(ctx.getPathPoints());
```

```
function ptsToRects(pts){
    ctx.fillStyle='red';
    var i=0;
    requestAnimationFrame(animate);
    function animate(){
        ctx.fillRect(pts[i].x-0.50,pts[i].y-0.50,tolerance,tolerance);
        i++;
        if(i<pts.length){ requestAnimationFrame(animate); }
    }
}


///////////////////////////////////////
// A Plug-in
///////////////////////////////////////

// Modify the Canvas' Context to calculate a set of approximately
//     evenly spaced waypoints as it draws path(s).
function plotPathCommands(ctx,sampleCount,pointSpacing){
    ctx.mySampleCount=sampleCount;
    ctx.myPointSpacing=pointSpacing;
    ctx.myTolerance=pointSpacing*pointSpacing;
    ctx.myBeginPath=ctx.beginPath;
    ctx.myMoveTo=ctx.moveTo;
    ctx.myLineTo=ctx.lineTo;
    ctx.myQuadraticCurveTo=ctx.quadraticCurveTo;
    ctx.myBezierCurveTo=ctx.bezierCurveTo;
    // don't use myPathPoints[] directly -- use "ctx.getPathPoints"
    ctx.myPathPoints=[];
    ctx.beginPath=function(){
        this.myLastX=0;
        this.myLastY=0;
        this.myBeginPath();
    }
    ctx.moveTo=function(x,y){
        this.myLastX=x;
        this.myLastY=y;
        this.myMoveTo(x,y);
    }
    ctx.lineTo=function(x,y){
        var pts=plotLine(this.myTolerance,this.myLastX,this.myLastY,x,y);
        Array.prototype.push.apply(this.myPathPoints,pts);
        this.myLastX=x;
        this.myLastY=y;
        this.myLineTo(x,y);
    }
    ctx.quadraticCurveTo=function(x0,y0,x1,y1){
        var
pts=plotQBez(this.mySampleCount,this.myTolerance,this.myLastX,this.myLastY,x0,y0,x1,y1);
        Array.prototype.push.apply(this.myPathPoints,pts);
        this.myLastX=x1;
        this.myLastY=y1;
        this.myQuadraticCurveTo(x0,y0,x1,y1);
    }
    ctx.bezierCurveTo=function(x0,y0,x1,y1,x2,y2){
        var
pts=plotCBez(this.mySampleCount,this.myTolerance,this.myLastX,this.myLastY,x0,y0,x1,y1,x2,y2);
        Array.prototype.push.apply(this.myPathPoints,pts);
        this.myLastX=x2;
        this.myLastY=y2;
        this.myBezierCurveTo(x0,y0,x1,y1,x2,y2);
    }
```

```
    ctx.getPathPoints=function(){
        return(this.myPathPoints.slice());
    }
    ctx.clearPathPoints=function(){
        this.myPathPoints.length=0;
    }
    ctx.stopPlottingPathCommands=function(){
        if(!this.myBeginPath){return;}
        this.beginPath=this.myBeginPath;
        this.moveTo=this.myMoveTo;
        this.lineTo=this.myLineTo;
        this.quadraticCurveto=this.myQuadraticCurveTo;
        this.bezierCurveTo=this.myBezierCurveTo;
        this.myBeginPath=undefined;
    }
}


////////////////////////////////
// Helper functions
////////////////////////////////

// Return: a set of approximately evenly spaced points along a cubic Bezier curve
//
// Attribution: Stackoverflow's @Blindman67
// Cite:
http://stackoverflow.com/questions/36637211/drawing-a-curved-line-in-css-or-canvas-and-moving-circle-
along-it/36827074#36827074
// As modified from the above citation
//
// ptCount: sample this many points at interval along the curve
// pxTolerance: approximate spacing allowed between points
// Ax,Ay,Bx,By,Cx,Cy,Dx,Dy: control points defining the curve
//
function plotCBez(ptCount,pxTolerance,Ax,Ay,Bx,By,Cx,Cy,Dx,Dy){
    var deltaBAx=Bx-Ax;
    var deltaCBx=Cx-Bx;
    var deltaDCx=Dx-Cx;
    var deltaBAy=By-Ay;
    var deltaCBy=Cy-By;
    var deltaDCy=Dy-Cy;
    var ax,ay,bx,by;
    var lastX=-10000;
    var lastY=-10000;
    var pts=[{x:Ax,y:Ay}];
    for(var i=1;i<ptCount;i++){
        var t=i/ptCount;
        ax=Ax+deltaBAx*t;
        bx=Bx+deltaCBx*t;
        cx=Cx+deltaDCx*t;
        ax+=(bx-ax)*t;
        bx+=(cx-bx)*t;
        //
        ay=Ay+deltaBAy*t;
        by=By+deltaCBy*t;
        cy=Cy+deltaDCy*t;
        ay+=(by-ay)*t;
        by+=(cy-by)*t;
        var x=ax+(bx-ax)*t;
        var y=ay+(by-ay)*t;
        var dx=x-lastX;
        var dy=y-lastY;
```

```
        if(dx*dx+dy*dy>pxTolerance){
            pts.push({x:x,y:y});
            lastX=x;
            lastY=y;
        }
    }
    pts.push({x:Dx,y:Dy});
    return(pts);
}

// Return: an array of approximately evenly spaced points along a Quadratic curve
//
// Attribution: Stackoverflow's @Blindman67
// Cite:
http://stackoverflow.com/questions/36637211/drawing-a-curved-line-in-css-or-canvas-and-moving-circle-along-it/36827074#36827074
// As modified from the above citation
//
// ptCount: sample this many points at interval along the curve
// pxTolerance: approximate spacing allowed between points
// Ax,Ay,Bx,By,Cx,Cy: control points defining the curve
//
function plotQBez(ptCount,pxTolerance,Ax,Ay,Bx,By,Cx,Cy){
    var deltaBAx=Bx-Ax;
    var deltaCBx=Cx-Bx;
    var deltaBAy=By-Ay;
    var deltaCBy=Cy-By;
    var ax,ay;
    var lastX=-10000;
    var lastY=-10000;
    var pts=[{x:Ax,y:Ay}];
    for(var i=1;i<ptCount;i++){
        var t=i/ptCount;
        ax=Ax+deltaBAx*t;
        ay=Ay+deltaBAy*t;
        var x=ax+((Bx+deltaCBx*t)-ax)*t;
        var y=ay+((By+deltaCBy*t)-ay)*t;
        var dx=x-lastX;
        var dy=y-lastY;
        if(dx*dx+dy*dy>pxTolerance){
            pts.push({x:x,y:y});
            lastX=x;
            lastY=y;
        }
    }
    pts.push({x:Cx,y:Cy});
    return(pts);
}

// Return: an array of approximately evenly spaced points along a line
//
// pxTolerance: approximate spacing allowed between points
// Ax,Ay,Bx,By: end points defining the line
//
function plotLine(pxTolerance,Ax,Ay,Bx,By){
    var dx=Bx-Ax;
    var dy=By-Ay;
    var ptCount=parseInt(Math.sqrt(dx*dx+dy*dy))*3;
    var lastX=-10000;
    var lastY=-10000;
    var pts=[{x:Ax,y:Ay}];
    for(var i=1;i<=ptCount;i++){
```

```
        var t=i/ptCount;
        var x=Ax+dx*t;
        var y=Ay+dy*t;
        var dx1=x-lastX;
        var dy1=y-lastY;
        if(dx1*dx1+dy1*dy1>pxTolerance){
            pts.push({x:x,y:y});
            lastX=x;
            lastY=y;
        }
    }
    pts.push({x:Bx,y:By});
    return(pts);
}
```

Section 7.7: Split bezier curves at position

This example splits cubic and bezier curves in two.

The function `splitCurveAt` splits the curve at `position` where `0.0` = start, `0.5` = middle, and `1` = end. It can split quadratic and cubic curves. The curve type is determined by the last x argument `x4`. If not **`undefined`** or **`null`** then it assumes the curve is cubic else the curve is a quadratic

Example usage

Splitting quadratic bezier curve in two

```
var p1 = {x : 10 , y : 100};
var p2 = {x : 100, y : 200};
var p3 = {x : 200, y : 0};
var newCurves = splitCurveAt(0.5, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y)

var i = 0;
var p = newCurves
// Draw the 2 new curves
// Assumes ctx is canvas 2d context
ctx.lineWidth = 1;
ctx.strokeStyle = "black";
ctx.beginPath();
ctx.moveTo(p[i++],p[i++]);
ctx.quadraticCurveTo(p[i++], p[i++], p[i++], p[i++]);
ctx.quadraticCurveTo(p[i++], p[i++], p[i++], p[i++]);
ctx.stroke();
```

Splitting cubic bezier curve in two

```
var p1 = {x : 10 , y : 100};
var p2 = {x : 100, y : 200};
var p3 = {x : 200, y : 0};
var p4 = {x : 300, y : 100};
var newCurves = splitCurveAt(0.5, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, p4.x, p4.y)

var i = 0;
var p = newCurves
// Draw the 2 new curves
// Assumes ctx is canvas 2d context
ctx.lineWidth = 1;
ctx.strokeStyle = "black";
ctx.beginPath();
```

```
ctx.moveTo(p[i++],p[i++]);
ctx.bezierCurveTo(p[i++], p[i++], p[i++], p[i++], p[i++], p[i++]);
ctx.bezierCurveTo(p[i++], p[i++], p[i++], p[i++], p[i++], p[i++]);
ctx.stroke();
```

The split function

splitCurveAt = function(position, x1, y1, x2, y2, x3, y3, [x4, y4])

> **Note:** Arguments inside [x4, y4] are optional.

> **Note:** The function has some optional commented `/* */` code that deals with edge cases where the resulting curves may have zero length, or fall outside the start or ends of the original curve. As is attempting to split a curve outside the valid range for `position >= 0` or `position >= 1` will throw a range error. This can be removed and will work just fine, though you may have resulting curves that have zero length.

```
// With throw RangeError if not 0 < position < 1
// x1, y1, x2, y2, x3, y3 for quadratic curves
// x1, y1, x2, y2, x3, y3, x4, y4 for cubic curves
// Returns an array of points representing 2 curves. The curves are the same type as the split curve
var splitCurveAt = function(position, x1, y1, x2, y2, x3, y3, x4, y4){
    var v1, v2, v3, v4, quad, retPoints, i, c;

    // =============================================================================================
    // you may remove this as the function will still work and resulting curves will still render
    // but other curve functions may not like curves with 0 length
    // =============================================================================================
    if(position <= 0 || position >= 1){
        throw RangeError("spliteCurveAt requires position > 0 && position < 1");
    }

    // =============================================================================================
    // If you remove the above range error you may use one or both of the following commented
sections
    // Splitting curves position < 0 or position > 1 will still create valid curves but they will
    // extend past the end points

    // =============================================================================================
    // Lock the position to split on the curve.
    /* optional A
    position = position < 0 ? 0 : position > 1 ? 1 : position;
    optional A end */

    // =============================================================================================
    // the next commented section will return the original curve if the split results in 0 length
curve
    // You may wish to uncomment this If you desire such functionality
    /*  optional B
    if(position <= 0 || position >= 1){
        if(x4 === undefined || x4 === null){
            return [x1, y1, x2, y2, x3, y3];
        }else{
            return [x1, y1, x2, y2, x3, y3, x4, y4];
        }
    }
```

```
    optional B end */


    retPoints = []; // array of coordinates
    i = 0;
    quad = false;  // presume cubic bezier
    v1 = {};
    v2 = {};
    v4 = {};
    v1.x = x1;
    v1.y = y1;
    v2.x = x2;
    v2.y = y2;
    if(x4 === undefined || x4 === null){
        quad = true;  // this is a quadratic bezier
        v4.x = x3;
        v4.y = y3;
    }else{
        v3 = {};
        v3.x = x3;
        v3.y = y3;
        v4.x = x4;
        v4.y = y4;
    }
    c = position;
    retPoints[i++] = v1.x;  // start point
    retPoints[i++] = v1.y;

    if(quad){ // split quadratic bezier
        retPoints[i++] = (v1.x += (v2.x - v1.x) * c);  // new control point for first curve
        retPoints[i++] = (v1.y += (v2.y - v1.y) * c);
        v2.x += (v4.x - v2.x) * c;
        v2.y += (v4.y - v2.y) * c;
        retPoints[i++] = v1.x + (v2.x - v1.x) * c;  // new end and start of first and second curves
        retPoints[i++] = v1.y + (v2.y - v1.y) * c;
        retPoints[i++] = v2.x;  // new control point for second curve
        retPoints[i++] = v2.y;
        retPoints[i++] = v4.x;  // new endpoint of second curve
        retPoints[i++] = v4.y;
        //=======================================================
        // return array with 2 curves
        return retPoints;
    }
    retPoints[i++] = (v1.x += (v2.x - v1.x) * c); // first curve first control point
    retPoints[i++] = (v1.y += (v2.y - v1.y) * c);
    v2.x += (v3.x - v2.x) * c;
    v2.y += (v3.y - v2.y) * c;
    v3.x += (v4.x - v3.x) * c;
    v3.y += (v4.y - v3.y) * c;
    retPoints[i++] = (v1.x += (v2.x - v1.x) * c); // first curve second control point
    retPoints[i++] = (v1.y += (v2.y - v1.y) * c);
    v2.x += (v3.x - v2.x) * c;
    v2.y += (v3.y - v2.y) * c;
    retPoints[i++] = v1.x + (v2.x - v1.x) * c; // end and start point of first second curves
    retPoints[i++] = v1.y + (v2.y - v1.y) * c;
    retPoints[i++] = v2.x;  // second curve first control point
    retPoints[i++] = v2.y;
    retPoints[i++] = v3.x;  // second curve second control point
    retPoints[i++] = v3.y;
    retPoints[i++] = v4.x;  // endpoint of second curve
    retPoints[i++] = v4.y;
    //=======================================================
```

```
    // return array with 2 curves
    return retPoints;
}
```

Section 7.8: Trim bezier curve

This example show you how to trim a bezier.

The function trimBezier trims the ends off of the curve returning the curve `fromPos` to `toPos`. `fromPos` and `toPos` are in the range 0 to 1 inclusive, It can trim quadratic and cubic curves. The curve type is determined by the last x argument `x4`. If not **`undefined`** or **`null`** then it assumes the curve is cubic else the curve is a quadratic

The trimmed curve is returned as an array of points. 6 points for quadratic curves and 8 for cubic curves.

Example Usage

Trimming a quadratic curve.

```
var p1 = {x : 10 , y : 100};
var p2 = {x : 100, y : 200};
var p3 = {x : 200, y : 0};
var newCurve = splitCurveAt(0.25, 0.75, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y)

var i = 0;
var p = newCurve
// Draw the trimmed curve
// Assumes ctx is canvas 2d context
ctx.lineWidth = 1;
ctx.strokeStyle = "black";
ctx.beginPath();
ctx.moveTo(p[i++],p[i++]);
ctx.quadraticCurveTo(p[i++], p[i++], p[i++], p[i++]);
ctx.stroke();
```

Trimming a cubic curve.

```
var p1 = {x : 10 , y : 100};
var p2 = {x : 100, y : 200};
var p3 = {x : 200, y : 0};
var p4 = {x : 300, y : 100};
var newCurve = splitCurveAt(0.25, 0.75, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, p4.x, p4.y)

var i = 0;
var p = newCurve
// Draw the trimmed curve
// Assumes ctx is canvas 2d context
ctx.lineWidth = 1;
ctx.strokeStyle = "black";
ctx.beginPath();
ctx.moveTo(p[i++],p[i++]);
ctx.bezierCurveTo(p[i++], p[i++], p[i++], p[i++], p[i++], p[i++]);
ctx.stroke();
```

Example Function

trimBezier = function(fromPos, toPos, x1, y1, x2, y2, x3, y3, [x4, y4])

> **Note:** Arguments inside [x4, y4] are optional.

> **Note:** This function requires the function in the example Split Bezier Curves At in this section

```
var trimBezier = function(fromPos, toPos, x1, y1, x2, y2, x3, y3, x4, y4){
    var quad, i, s, retBez;
    quad = false;
    if(x4 === undefined || x4 === null){
        quad = true;  // this is a quadratic bezier
    }
    if(fromPos > toPos){ // swap is from is after to
        i = fromPos;
        fromPos = toPos
        toPos = i;
    }
    // clamp to on the curve
    toPos = toPos <= 0 ? 0 : toPos >= 1 ? 1 : toPos;
    fromPos = fromPos <= 0 ? 0 : fromPos >= 1 ? 1 : fromPos;
    if(toPos === fromPos){
        s = splitBezierAt(toPos, x1, y1, x2, y2, x3, y3, x4, y4);
        i = quad ? 4 : 6;
        retBez = [s[i], s[i+1], s[i], s[i+1], s[i], s[i+1]];
        if(!quad){
            retBez.push(s[i], s[i+1]);
        }
        return retBez;
    }
    if(toPos === 1 && fromPos === 0){       // no trimming required
        retBez = [x1, y1, x2, y2, x3, y3];  // return original bezier
        if(!quad){
            retBez.push(x4, y4);
        }
        return retBez;
    }
    if(fromPos === 0){
        if(toPos < 1){
            s = splitBezierAt(toPos, x1, y1, x2, y2, x3, y3, x4, y4);
            i = 0;
            retBez = [s[i++], s[i++], s[i++], s[i++], s[i++], s[i++]];
            if(!quad){
                retBez.push(s[i++], s[i++]);
            }
        }
        return retBez;
    }
    if(toPos === 1){
        if(fromPos < 1){
            s = splitBezierAt(toPos, x1, y1, x2, y2, x3, y3, x4, y4);
            i = quad ? 4 : 6;
            retBez = [s[i++], s[i++], s[i++], s[i++], s[i++], s[i++]];
            if(!quad){
                retBez.push(s[i++], s[i++]);
            }
        }
        return retBez;
    }
    s = splitBezierAt(fromPos, x1, y1, x2, y2, x3, y3, x4, y4);
    if(quad){
        i = 4;
```

```
        toPos = (toPos - fromPos) / (1 - fromPos);
        s = splitBezierAt(toPos, s[i++], s[i++], s[i++], s[i++], s[i++], s[i++]);
        i = 0;
        retBez = [s[i++], s[i++], s[i++], s[i++], s[i++], s[i++]];
        return retBez;

    }
    i = 6;
    toPos = (toPos - fromPos) / (1 - fromPos);
    s = splitBezierAt(toPos, s[i++], s[i++], s[i++], s[i++], s[i++], s[i++], s[i++], s[i++]);
    i = 0;
    retBez = [s[i++], s[i++], s[i++], s[i++], s[i++], s[i++], s[i++], s[i++]];
    return retBez;
}
```

Section 7.9: Length of a Cubic Bezier Curve (a close approximation)

Given the 4 points of a cubic Bezier curve the following function returns its length.

Method: The length of a cubic Bezier curve does not have a direct mathematical calculation. This "brute force" method finds a sampling of points along the curve and calculates the total distance spanned by those points.

Accuracy: The approximate length is 99+% accurate using the default sampling size of 40.

```
// Return: Close approximation of the length of a Cubic Bezier curve
//
// Ax,Ay,Bx,By,Cx,Cy,Dx,Dy: the 4 control points of the curve
// sampleCount [optional, default=40]: how many intervals to calculate
// Requires: cubicQxy (included below)
//
function cubicBezierLength(Ax,Ay,Bx,By,Cx,Cy,Dx,Dy,sampleCount){
    var ptCount=sampleCount||40;
    var totDist=0;
    var lastX=Ax;
    var lastY=Ay;
    var dx,dy;
    for(var i=1;i<ptCount;i++){
        var pt=cubicQxy(i/ptCount,Ax,Ay,Bx,By,Cx,Cy,Dx,Dy);
        dx=pt.x-lastX;
        dy=pt.y-lastY;
        totDist+=Math.sqrt(dx*dx+dy*dy);
        lastX=pt.x;
        lastY=pt.y;
    }
    dx=Dx-lastX;
    dy=Dy-lastY;
    totDist+=Math.sqrt(dx*dx+dy*dy);
    return(parseInt(totDist));
}


// Return: an [x,y] point along a cubic Bezier curve at interval T
//
// Attribution: Stackoverflow's @Blindman67
// Cite:
http://stackoverflow.com/questions/36637211/drawing-a-curved-line-in-css-or-canvas-and-moving-circle-along-it/36827074#36827074
```

```
// As modified from the above citation
//
// t: an interval along the curve (0<=t<=1)
// ax,ay,bx,by,cx,cy,dx,dy: control points defining the curve
//
function cubicQxy(t,ax,ay,bx,by,cx,cy,dx,dy) {
    ax += (bx - ax) * t;
    bx += (cx - bx) * t;
    cx += (dx - cx) * t;
    ax += (bx - ax) * t;
    bx += (cx - bx) * t;
    ay += (by - ay) * t;
    by += (cy - by) * t;
    cy += (dy - cy) * t;
    ay += (by - ay) * t;
    by += (cy - by) * t;
    return({
        x:ax +(bx - ax) * t,
        y:ay +(by - ay) * t
    });
}
```

Section 7.10: Length of a Quadratic Curve

Given the 3 points of a quadratic curve the following function returns the length.

```
function quadraticBezierLength(x1,y1,x2,y2,x3,y3)
    var a, e, c, d, u, a1, e1, c1, d1, u1, v1x, v1y;

    v1x = x2 * 2;
    v1y = y2 * 2;
    d = x1 - v1x + x3;
    d1 = y1 - v1y + y3;
    e = v1x - 2 * x1;
    e1 = v1y - 2 * y1;
    c1 = (a = 4 * (d * d + d1 * d1));
    c1 += (b = 4 * (d * e + d1 * e1));
    c1 += (c = e * e + e1 * e1);
    c1 = 2 * Math.sqrt(c1);
    a1 = 2 * a * (u = Math.sqrt(a));
    u1 = b / u;
    a = 4 * c * a - b * b;
    c = 2 * Math.sqrt(c);
    return (a1 * c1 + u * b * (c1 - c) + a * Math.log((2 * u + u1 + c1) / (u1 + c))) / (4 * a1);
}
```

Derived from the quadratic bezier function F(t) = a * (1 - t)2 + 2 * b * (1 - t) * t + c * t2

Chapter 8: Dragging Path Shapes & Images on Canvas

Section 8.1: How shapes & images REALLY(!) "move" on the Canvas

A problem: Canvas only remembers pixels, not shapes or images

This is an image of a circular beach ball, and of course, you can't drag the ball around the image.



It may surprise you that just like an image, if you draw a circle on a Canvas you cannot drag that circle around the canvas. That's because the canvas won't remember where it drew the circle.

```
// this arc (==circle) is not draggable!!
context.beginPath();
context.arc(20, 30, 15, 0, Math.PI*2);
context.fillStyle='blue';
context.fill();
```

What the Canvas DOESN'T know...

- ...where you drew the circle (it does not know x,y =[20,30]).
- ...the size of the circle (it does not know radius=15).
- ...the color of the circle. (it does not know the circle is blue).

What the Canvas DOES know...

Canvas knows the color of every pixel on it's drawing surface.

The canvas can tell you that at x,y==[20,30] there is a blue pixel, but it does not know if this blue pixel is part of a circle.

What this means...

This means everything drawn on the Canvas is permanent: immovable and unchangeable.

- Canvas can't move the circle or resize the circle.
- Canvas can't recolor the circle or erase the circle.
- Canvas can't say if the mouse is hovering over the circle.
- Canvas can't say if the circle is colliding with another circle.
- Canvas can't let a user drag the circle around the Canvas.

But Canvas can give the I-L-L-U-S-I-O-N of movement

Canvas can give the **illusion of movement** by continuously erasing the circle and redrawing it in a different position. By redrawing the Canvas many times per second, the eye is fooled into seeing the circle move across the Canvas.

- Erase the canvas
- Update the circle's position
- Redraw the circle in it's new position
- Repeat, repeat, repeat ...

This code gives the **illusion of movement** by continuously redrawing a circle in new positions.

```
// create a canvas
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
ctx.fillStyle='red';
document.body.appendChild(canvas);

// a variable indicating a circle's X position
var circleX=20;

// start animating the circle across the canvas
// by continuously erasing & redrawing the circle
// in new positions
requestAnimationFrame(animate);

function animate(){
    // update the X position of the circle
    circleX++;
    // redraw the circle in it's new position
    ctx.clearRect(0,0,canvas.width,canvas.height);
    ctx.beginPath();
    ctx.arc( circleX, 30,15,0,Math.PI*2 );
    ctx.fill();
    // request another animate() loop
    requestAnimationFrame(animate);
}
```

Section 8.2: Dragging circles & rectangles around the Canvas

What is a "Shape"?

You typically save your shapes by creating a JavaScript "shape" object representing each shape.

```
var myCircle = { x:30, y:20, radius:15 };
```

Of course, you're not really saving shapes. Instead, you're saving the definition of how to draw the shapes.

Then put every shape-object into an array for easy reference.

```
// save relevant information about shapes drawn on the canvas
var shapes=[];

// define one circle and save it in the shapes[] array
```

```
shapes.push( {x:10, y:20, radius:15, fillcolor:'blue'} );

// define one rectangle and save it in the shapes[] array
shapes.push( {x:10, y:100, width:50, height:35, fillcolor:'red'} );
```

Using mouse events to do Dragging

Dragging a shape or image requires responding to these mouse events:

On mousedown:

Test if any shape is under the mouse. If a shape is under the mouse, the user is intending to drag that shape. So keep a reference to that shape and set a true/false `isDragging` flag indicating that a drag is in process.

On mousemove:

Calculate the distance that the mouse has been dragged since the last `mousemove` event and change the dragged shape's position by that distance. To change the shape's position, you change the `x,y` position properties in that shape's object.

On mouseup or mouseout:

The user is intending to stop the drag operation, so clear the "isDragging" flag. Dragging is completed.

Demo: Dragging circles & rectangles on the canvas

This demo drags circles & rectangles on the canvas by responding to mouse events and giving the illusion of movement by clearing and redrawing.

```
// canvas related vars
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
var cw=canvas.width;
var ch=canvas.height;
document.body.appendChild(canvas);
canvas.style.border='1px solid red';

// used to calc canvas position relative to window
function reOffset(){
    var BB=canvas.getBoundingClientRect();
    offsetX=BB.left;
    offsetY=BB.top;
}
var offsetX,offsetY;
reOffset();
window.onscroll=function(e){ reOffset(); }
window.onresize=function(e){ reOffset(); }
canvas.onresize=function(e){ reOffset(); }

// save relevant information about shapes drawn on the canvas
var shapes=[];
// define one circle and save it in the shapes[] array
shapes.push( {x:30, y:30, radius:15, color:'blue'} );
// define one rectangle and save it in the shapes[] array
shapes.push( {x:100, y:-1, width:75, height:35, color:'red'} );

// drag related vars
var isDragging=false;
var startX,startY;
```

```
// hold the index of the shape being dragged (if any)
var selectedShapeIndex;

// draw the shapes on the canvas
drawAll();

// listen for mouse events
canvas.onmousedown=handleMouseDown;
canvas.onmousemove=handleMouseMove;
canvas.onmouseup=handleMouseUp;
canvas.onmouseout=handleMouseOut;

// given mouse X & Y (mx & my) and shape object
// return true/false whether mouse is inside the shape
function isMouseInShape(mx,my,shape){
    if(shape.radius){
        // this is a circle
        var dx=mx-shape.x;
        var dy=my-shape.y;
        // math test to see if mouse is inside circle
        if(dx*dx+dy*dy<shape.radius*shape.radius){
            // yes, mouse is inside this circle
            return(true);
        }
    }else if(shape.width){
        // this is a rectangle
        var rLeft=shape.x;
        var rRight=shape.x+shape.width;
        var rTop=shape.y;
        var rBott=shape.y+shape.height;
        // math test to see if mouse is inside rectangle
        if( mx>rLeft && mx<rRight && my>rTop && my<rBott){
            return(true);
        }
    }
    // the mouse isn't in any of the shapes
    return(false);
}

function handleMouseDown(e){
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // calculate the current mouse position
    startX=parseInt(e.clientX-offsetX);
    startY=parseInt(e.clientY-offsetY);
    // test mouse position against all shapes
    // post result if mouse is in a shape
    for(var i=0;i<shapes.length;i++){
        if(isMouseInShape(startX,startY,shapes[i])){
            // the mouse is inside this shape
            // select this shape
            selectedShapeIndex=i;
            // set the isDragging flag
            isDragging=true;
            // and return (==stop looking for
            //     further shapes under the mouse)
            return;
        }
    }
}
```

```
function handleMouseUp(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // the drag is over -- clear the isDragging flag
    isDragging=false;
}

function handleMouseOut(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // the drag is over -- clear the isDragging flag
    isDragging=false;
}

function handleMouseMove(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // calculate the current mouse position
    mouseX=parseInt(e.clientX-offsetX);
    mouseY=parseInt(e.clientY-offsetY);
    // how far has the mouse dragged from its previous mousemove position?
    var dx=mouseX-startX;
    var dy=mouseY-startY;
    // move the selected shape by the drag distance
    var selectedShape=shapes[selectedShapeIndex];
    selectedShape.x+=dx;
    selectedShape.y+=dy;
    // clear the canvas and redraw all shapes
    drawAll();
    // update the starting drag position (== the current mouse position)
    startX=mouseX;
    startY=mouseY;
}

// clear the canvas and
// redraw all shapes in their current positions
function drawAll(){
    ctx.clearRect(0,0,cw,ch);
    for(var i=0;i<shapes.length;i++){
        var shape=shapes[i];
        if(shape.radius){
            // it's a circle
            ctx.beginPath();
            ctx.arc(shape.x,shape.y,shape.radius,0,Math.PI*2);
            ctx.fillStyle=shape.color;
            ctx.fill();
        }else if(shape.width){
            // it's a rectangle
            ctx.fillStyle=shape.color;
            ctx.fillRect(shape.x,shape.y,shape.width,shape.height);
        }
    }
}
```

Section 8.3: Dragging irregular shapes around the Canvas

Most Canvas drawings are either rectangular (rectangles, images, text-blocks) or circular (circles).

Circles & rectangles have mathematical tests to check if the mouse is inside them. This makes testing circles and rectangles easy, quick and efficient. You can "hit-test" hundreds of circles or rectangles in a fraction of a second.

You can also drag irregular shapes. But irregular shapes have no quick mathematical hit-test. Fortunately, irregular shapes do have a built-in hit-test to determine if a point (mouse) is inside the shape: `context.isPointInPath`. While `isPointInPath` works well, it is not nearly as efficient as purely mathematical hit-tests -- it is often up to 10X slower than pure mathematical hit-tests.

One requirement when using `isPointInPath` is that you must "redefine" the Path being tested immediately before calling `isPointInPath`. "Redefine" means you must issue the path drawing commands (as above), but you don't need to stroke() or fill() the Path before testing it with `isPointInPath`. This way you can test previously drawn Paths without having to overwrite (stroke/fill) those previous Paths on the Canvas itself.

The irregular shape doesn't need to be as common as the everyday triangle. You can also hit-test any wildly irregular Paths.

This annotated example shows how to drag irregular Path shapes as well as circles and rectangles:

```
// canvas related vars
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
var cw=canvas.width;
var ch=canvas.height;
document.body.appendChild(canvas);
canvas.style.border='1px solid red';

// used to calc canvas position relative to window
function reOffset(){
    var BB=canvas.getBoundingClientRect();
    offsetX=BB.left;
    offsetY=BB.top;
}
var offsetX,offsetY;
reOffset();
window.onscroll=function(e){ reOffset(); }
window.onresize=function(e){ reOffset(); }
canvas.onresize=function(e){ reOffset(); }

// save relevant information about shapes drawn on the canvas
var shapes=[];
// define one circle and save it in the shapes[] array
shapes.push( {x:20, y:20, radius:15, color:'blue'} );
// define one rectangle and save it in the shapes[] array
shapes.push( {x:100, y:-1, width:75, height:35, color:'red'} );
// define one triangle path and save it in the shapes[] array
shapes.push( {x:0, y:0, points:[{x:50,y:30},{x:75,y:60},{x:25,y:60}],color:'green'} );

// drag related vars
var isDragging=false;
var startX,startY;

// hold the index of the shape being dragged (if any)
var selectedShapeIndex;

// draw the shapes on the canvas
```

```
drawAll();

// listen for mouse events
canvas.onmousedown=handleMouseDown;
canvas.onmousemove=handleMouseMove;
canvas.onmouseup=handleMouseUp;
canvas.onmouseout=handleMouseOut;

// given mouse X & Y (mx & my) and shape object
// return true/false whether mouse is inside the shape
function isMouseInShape(mx,my,shape){
    if(shape.radius){
        // this is a circle
        var dx=mx-shape.x;
        var dy=my-shape.y;
        // math test to see if mouse is inside circle
        if(dx*dx+dy*dy<shape.radius*shape.radius){
            // yes, mouse is inside this circle
            return(true);
        }
    }else if(shape.width){
        // this is a rectangle
        var rLeft=shape.x;
        var rRight=shape.x+shape.width;
        var rTop=shape.y;
        var rBott=shape.y+shape.height;
        // math test to see if mouse is inside rectangle
        if( mx>rLeft && mx<rRight && my>rTop && my<rBott){
            return(true);
        }
    }else if(shape.points){
        // this is a polyline path
        // First redefine the path again (no need to stroke/fill!)
        defineIrregularPath(shape);
        // Then hit-test with isPointInPath
        if(ctx.isPointInPath(mx,my)){
            return(true);
        }
    }
    // the mouse isn't in any of the shapes
    return(false);
}

function handleMouseDown(e){
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // calculate the current mouse position
    startX=parseInt(e.clientX-offsetX);
    startY=parseInt(e.clientY-offsetY);
    // test mouse position against all shapes
    // post result if mouse is in a shape
    for(var i=0;i<shapes.length;i++){
        if(isMouseInShape(startX,startY,shapes[i])){
            // the mouse is inside this shape
            // select this shape
            selectedShapeIndex=i;
            // set the isDragging flag
            isDragging=true;
            // and return (==stop looking for
            //     further shapes under the mouse)
            return;
```

```
        }
    }
}

function handleMouseUp(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // the drag is over -- clear the isDragging flag
    isDragging=false;
}

function handleMouseOut(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // the drag is over -- clear the isDragging flag
    isDragging=false;
}

function handleMouseMove(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // calculate the current mouse position
    mouseX=parseInt(e.clientX-offsetX);
    mouseY=parseInt(e.clientY-offsetY);
    // how far has the mouse dragged from its previous mousemove position?
    var dx=mouseX-startX;
    var dy=mouseY-startY;
    // move the selected shape by the drag distance
    var selectedShape=shapes[selectedShapeIndex];
    selectedShape.x+=dx;
    selectedShape.y+=dy;
    // clear the canvas and redraw all shapes
    drawAll();
    // update the starting drag position (== the current mouse position)
    startX=mouseX;
    startY=mouseY;
}

// clear the canvas and
// redraw all shapes in their current positions
function drawAll(){
    ctx.clearRect(0,0,cw,ch);
    for(var i=0;i<shapes.length;i++){
        var shape=shapes[i];
        if(shape.radius){
            // it's a circle
            ctx.beginPath();
            ctx.arc(shape.x,shape.y,shape.radius,0,Math.PI*2);
            ctx.fillStyle=shape.color;
            ctx.fill();
        }else if(shape.width){
            // it's a rectangle
            ctx.fillStyle=shape.color;
```

```
            ctx.fillRect(shape.x,shape.y,shape.width,shape.height);
        }else if(shape.points){
            // its a polyline path
            defineIrregularPath(shape);
            ctx.fillStyle=shape.color;
            ctx.fill();
        }
    }
}

function defineIrregularPath(shape){
    var points=shape.points;
    ctx.beginPath();
    ctx.moveTo(shape.x+points[0].x,shape.y+points[0].y);
    for(var i=1;i<points.length;i++){
        ctx.lineTo(shape.x+points[i].x,shape.y+points[i].y);
    }
    ctx.closePath();
}
```

Section 8.4: Dragging images around the Canvas

See this Example for a general explanation of dragging Shapes around the Canvas.

This annotated example shows how to drag images around the Canvas

```
// canvas related vars
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
canvas.width=378;
canvas.height=378;
var cw=canvas.width;
var ch=canvas.height;
document.body.appendChild(canvas);
canvas.style.border='1px solid red';

// used to calc canvas position relative to window
function reOffset(){
    var BB=canvas.getBoundingClientRect();
    offsetX=BB.left;
    offsetY=BB.top;
}
var offsetX,offsetY;
reOffset();
window.onscroll=function(e){ reOffset(); }
window.onresize=function(e){ reOffset(); }
canvas.onresize=function(e){ reOffset(); }

// save relevant information about shapes drawn on the canvas
var shapes=[];

// drag related vars
var isDragging=false;
var startX,startY;

// hold the index of the shape being dragged (if any)
var selectedShapeIndex;

// load the image
var card=new Image();
card.onload=function(){
```

```
    // define one image and save it in the shapes[] array
    shapes.push( {x:30, y:10, width:127, height:150, image:card} );
    // draw the shapes on the canvas
    drawAll();
    // listen for mouse events
    canvas.onmousedown=handleMouseDown;
    canvas.onmousemove=handleMouseMove;
    canvas.onmouseup=handleMouseUp;
    canvas.onmouseout=handleMouseOut;
};
// put your image src here!
card.src='https://dl.dropboxusercontent.com/u/139992952/stackoverflow/card.png';


// given mouse X & Y (mx & my) and shape object
// return true/false whether mouse is inside the shape
function isMouseInShape(mx,my,shape){
    // is this shape an image?
    if(shape.image){
        // this is a rectangle
        var rLeft=shape.x;
        var rRight=shape.x+shape.width;
        var rTop=shape.y;
        var rBott=shape.y+shape.height;
        // math test to see if mouse is inside image
        if( mx>rLeft && mx<rRight && my>rTop && my<rBott){
            return(true);
        }
    }
    // the mouse isn't in any of this shapes
    return(false);
}

function handleMouseDown(e){
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // calculate the current mouse position
    startX=parseInt(e.clientX-offsetX);
    startY=parseInt(e.clientY-offsetY);
    // test mouse position against all shapes
    // post result if mouse is in a shape
    for(var i=0;i<shapes.length;i++){
        if(isMouseInShape(startX,startY,shapes[i])){
            // the mouse is inside this shape
            // select this shape
            selectedShapeIndex=i;
            // set the isDragging flag
            isDragging=true;
            // and return (==stop looking for
            //     further shapes under the mouse)
            return;
        }
    }
}

function handleMouseUp(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
```

```
    // the drag is over -- clear the isDragging flag
    isDragging=false;
}

function handleMouseOut(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // the drag is over -- clear the isDragging flag
    isDragging=false;
}

function handleMouseMove(e){
    // return if we're not dragging
    if(!isDragging){return;}
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
    // calculate the current mouse position
    mouseX=parseInt(e.clientX-offsetX);
    mouseY=parseInt(e.clientY-offsetY);
    // how far has the mouse dragged from its previous mousemove position?
    var dx=mouseX-startX;
    var dy=mouseY-startY;
    // move the selected shape by the drag distance
    var selectedShape=shapes[selectedShapeIndex];
    selectedShape.x+=dx;
    selectedShape.y+=dy;
    // clear the canvas and redraw all shapes
    drawAll();
    // update the starting drag position (== the current mouse position)
    startX=mouseX;
    startY=mouseY;
}

// clear the canvas and
// redraw all shapes in their current positions
function drawAll(){
    ctx.clearRect(0,0,cw,ch);
    for(var i=0;i<shapes.length;i++){
        var shape=shapes[i];
        if(shape.image){
            // it's an image
            ctx.drawImage(shape.image,shape.x,shape.y);
        }
    }
}
```

Chapter 9: Media types and the canvas

Section 9.1: Basic loading and playing a video on the canvas

The canvas can be used to display video from a variety of sources. This example shows how to load a video as a file resource, display it and add a simple click on screen play/pause toggle.

This stackoverflow self answered question [How do I display a video using HTML5 canvas tag] shows the following example code in action.

Just an image

A video is just an image as far as the canvas is concerned. You can draw it like any image. The difference being the video can play and has sound.

Get canvas and basic setup

```
// It is assumed you know how to add a canvas and correctly size it.
var canvas = document.getElementById("myCanvas"); // get the canvas from the page
var ctx = canvas.getContext("2d");
var videoContainer; // object to hold video and associated info
```

Creating and loading the video

```
var video = document.createElement("video"); // create a video element
video.src = "urlOffVideo.webm";
// the video will now begin to load.
// As some additional info is needed we will place the video in a
// containing object for convenience
video.autoPlay = false; // ensure that the video does not auto play
video.loop = true; // set the video to loop.
videoContainer = {  // we will add properties as needed
     video : video,
     ready : false,
};
```

Unlike images elements videos don't have to be fully loaded to be displayed on the canvas. Videos also provide a host of extra events that can be used to monitor status of the video.

In this case we wish to know when the video is ready to play. `oncanplay` means that enough of the video has loaded to play some of it, but there may not be enough to play to the end.

```
video.oncanplay = readyToPlayVideo; // set the event to the play function that
                                   // can be found below
```

Alternatively you can use `oncanplaythrough` which will fire when enough of the video has loaded so that it can be played to the end.

```
video.oncanplaythrough = readyToPlayVideo; // set the event to the play function that
                                          // can be found below
```

Only use one of the canPlay events not both.

The can play event (equivalent to image onload)

```
function readyToPlayVideo(event){ // this is a reference to the video
    // the video may not match the canvas size so find a scale to fit
    videoContainer.scale = Math.min(
                         canvas.width / this.videoWidth,
```

```
                        canvas.height / this.videoHeight);
    videoContainer.ready = true;
    // the video can be played so hand it off to the display function
    requestAnimationFrame(undateCanvas);
}
```

Displaying

The video will not play itself on the canvas. You need to draw it for every new frame. As it is difficult to know the exact frame rate and when they occur the best approch is to display the video as if running at 60fps. If the frame rate is lower then w just render the same frame twice. If the frame rate is higher then there is nothing that can be don to see the extra frames so we just ignore them.

The video element is just a image element and can be draw like any image, you can scale, rotate, pan the video, mirror it, fade it, clip it and display only parts, draw it twice the second time with a global composite mode to add FX like lighten, screen, etc..

```
function updateCanvas(){
    ctx.clearRect(0,0,canvas.width,canvas.height); // Though not always needed
                                                     // you may get bad pixels from
                                                     // previous videos so clear to be
                                                     // safe
    // only draw if loaded and ready
    if(videoContainer !== undefined && videoContainer.ready){
        // find the top left of the video on the canvas
        var scale = videoContainer.scale;
        var vidH = videoContainer.video.videoHeight;
        var vidW = videoContainer.video.videoWidth;
        var top = canvas.height / 2 - (vidH /2 ) * scale;
        var left = canvas.width / 2 - (vidW /2 ) * scale;
        // now just draw the video the correct size
        ctx.drawImage(videoContainer.video, left, top, vidW * scale, vidH * scale);
        if(videoContainer.video.paused){ // if not playing show the paused screen
            drawPayIcon();
        }
    }
    // all done for display
    // request the next frame in 1/60th of a second
    requestAnimationFrame(updateCanvas);
}
```

Basic play pause control

Now we have the video loaded and displayed all we need is the play control. We will make it as a click toggle play on the screen. When the video is playing and the user clicks the video is paused. When paused the click resumes play. We will add a function to darken the video and draw an play icon (triangle)

```
function drawPayIcon(){
     ctx.fillStyle = "black";  // darken display
     ctx.globalAlpha = 0.5;
     ctx.fillRect(0,0,canvas.width,canvas.height);
     ctx.fillStyle = "#DDD"; // colour of play icon
     ctx.globalAlpha = 0.75; // partly transparent
     ctx.beginPath(); // create the path for the icon
     var size = (canvas.height / 2) * 0.5;  // the size of the icon
     ctx.moveTo(canvas.width/2 + size/2, canvas.height / 2); // start at the pointy end
     ctx.lineTo(canvas.width/2 - size/2, canvas.height / 2 + size);
     ctx.lineTo(canvas.width/2 - size/2, canvas.height / 2 - size);
```

```
        ctx.closePath();
        ctx.fill();
        ctx.globalAlpha = 1; // restore alpha
}
```

Now the play pause event

```
function playPauseClick(){
        if(videoContainer !== undefined && videoContainer.ready){
            if(videoContainer.video.paused){
                    videoContainer.video.play();
            }else{
                    videoContainer.video.pause();
            }
        }
}
// register the event
canvas.addEventListener("click",playPauseClick);
```

Summary

Playing a video is very easy using the canvas, adding effect in real time is also easy. There are however some limitations on formats, how you can play and seek. MDN HTMLMediaElement is the place to get the full reference to the video object.

Once the image has been drawn on the canvas you can use `ctx.getImageData` to access the pixels it contains. Or you can use `canvas.toDataURL` to snap a still and download it. (Only if the video is from a trusted source and does not taint the canvas).

Note if the video has sound then playing it will also play the sound.

Happy videoing.

Section 9.2: Capture canvas and Save as webM video

Creating a WebM video from canvas frames and playing in canvas, or upload, or downloading.

Example capture and play canvas

```
name = "CanvasCapture"; // Placed into the Mux and Write Application Name fields of the WebM header
quality = 0.7; // good quality 1 Best < 0.7 ok to poor
fps = 30; // I have tried all sorts of frame rates and all seem to work
          // Do some test to workout what your machine can handle as there
          // is a lot of variation between machines.
var video = new Groover.Video(fps,quality,name)
function capture(){
    if(video.timecode < 5000){ // 5 seconds
         setTimeout(capture,video.frameDelay);
    }else{
         var videoElement = document.createElement("video");
         videoElement.src = URL.createObjectURL(video.toBlob());
         document.body.appendChild(videoElement);
         video = undefined; // DeReference as it is memory hungry.
         return;
    }
    // first frame sets the video size
    video.addFrame(canvas); // Add current canvas frame
}
capture(); // start capture
```

Rather than put in a huge effort only to be rejected, this is a quick insert to see if acceptable. Will Give full details if accepted. Also include additional capture options for better HD capture rates (removed from this version, Can capture HD 1080 at 50fps on good machines.)

This was inspired by Wammy but is a complete rewrite with encode as you go methodology, greatly reducing the memory required during capture. Can capture more than 30 seconds better data, handling algorithms.

> **Note** frames are encoded into webP images. Only Chrome supports webP canvas encoding. For other browsers (Firefox and Edge) you will need to use a 3rd party webP encoder such as Libwebp Javascript Encoding WebP images via Javascript is slow. (will include addition of raw webp images support if accepted).

The webM encoder inspired by Whammy: A Real Time Javascript WebM

```
var Groover = (function(){
    // ensure webp is supported
    function canEncode(){
        var canvas = document.createElement("canvas");
        canvas.width = 8;
        canvas.height = 8;
        return canvas.toDataURL("image/webp",0.1).indexOf("image/webp") > -1;
    }
    if(!canEncode()){
        return undefined;
    }
    var webmData = null;
    var clusterTimecode = 0;
    var clusterCounter = 0;
    var CLUSTER_MAX_DURATION = 30000;
    var frameNumber = 0;
    var width;
    var height;
    var frameDelay;
    var quality;
    var name;
    const videoMimeType = "video/webm"; // the only one.
    const frameMimeType = 'image/webp'; // can be no other
    const S = String.fromCharCode;
    const dataTypes = {
        object : function(data){ return toBlob(data);},
        number : function(data){ return stream.num(data);},
        string : function(data){ return stream.str(data);},
        array  : function(data){ return data;},
        double2Str : function(num){
            var c = new Uint8Array((new Float64Array([num])).buffer);
            return S(c[7]) + S(c[6]) + S(c[5]) + S(c[4]) + S(c[3]) + S(c[2]) + S(c[1]) + S(c[0]);
        }
    };

    const stream = {
        num : function(num){ // writes int
            var parts = [];
            while(num > 0){ parts.push(num & 0xff); num = num >> 8; }
            return new Uint8Array(parts.reverse());
        },
        str : function(str){ // writes string
            var i, len, arr;
            len = str.length;
```

```
            arr = new Uint8Array(len);
            for(i = 0; i < len; i++){arr[i] = str.charCodeAt(i);}
            return arr;
        },
        compInt : function(num){ // could not find full details so bit of a guess
            if(num < 128){       // number is prefixed with a bit (1000 is on byte 0100 two, 0010
three and so on)
                num += 0x80;
                return new Uint8Array([num]);
            }else
            if(num < 0x4000){
                num += 0x4000;
                return new Uint8Array([num>>8, num])
            }else
            if(num < 0x200000){
                num += 0x200000;
                return new Uint8Array([num>>16, num>>8, num])
            }else
            if(num < 0x10000000){
                num += 0x10000000;
                return new Uint8Array([num>>24, num>>16, num>>8, num])
            }
        }
    }
    const ids = { // header names and values
        videoData          : 0x1a45dfa3,
        Version            : 0x4286,
        ReadVersion        : 0x42f7,
        MaxIDLength        : 0x42f2,
        MaxSizeLength      : 0x42f3,
        DocType            : 0x4282,
        DocTypeVersion     : 0x4287,
        DocTypeReadVersion : 0x4285,
        Segment            : 0x18538067,
        Info               : 0x1549a966,
        TimecodeScale      : 0x2ad7b1,
        MuxingApp          : 0x4d80,
        WritingApp         : 0x5741,
        Duration           : 0x4489,
        Tracks             : 0x1654ae6b,
        TrackEntry         : 0xae,
        TrackNumber        : 0xd7,
        TrackUID           : 0x63c5,
        FlagLacing         : 0x9c,
        Language           : 0x22b59c,
        CodecID            : 0x86,
        CodecName          : 0x258688,
        TrackType          : 0x83,
        Video              : 0xe0,
        PixelWidth         : 0xb0,
        PixelHeight        : 0xba,
        Cluster            : 0x1f43b675,
        Timecode           : 0xe7,
        Frame              : 0xa3,
        Keyframe           : 0x9d012a,
        FrameBlock         : 0x81,
    };
    const keyframeD64Header = '\x9d\x01\x2a'; //VP8 keyframe header 0x9d012a
    const videoDataPos = 1; // data pos of frame data header
    const defaultDelay = dataTypes.double2Str(1000/25);
    const header = [  // structure of webM header/chunks what ever they are called.
        ids.videoData,[
```

```
            ids.Version, 1,
            ids.ReadVersion, 1,
            ids.MaxIDLength, 4,
            ids.MaxSizeLength, 8,
            ids.DocType, 'webm',
            ids.DocTypeVersion, 2,
            ids.DocTypeReadVersion, 2
        ],
        ids.Segment, [
            ids.Info, [
                ids.TimecodeScale, 1000000,
                ids.MuxingApp, 'Groover',
                ids.WritingApp, 'Groover',
                ids.Duration, 0
            ],
            ids.Tracks,[
                ids.TrackEntry,[
                    ids.TrackNumber, 1,
                    ids.TrackUID, 1,
                    ids.FlagLacing, 0,     // always o
                    ids.Language, 'und',   // undefined I think this means
                    ids.CodecID, 'V_VP8',  // These I think must not change
                    ids.CodecName, 'VP8',  // These I think must not change
                    ids.TrackType, 1,
                    ids.Video, [
                        ids.PixelWidth, 0,
                        ids.PixelHeight, 0
                    ]
                ]
            ]
        ]
    ];
    function getHeader(){
        header[3][2][3] = name;
        header[3][2][5] = name;
        header[3][2][7] =  dataTypes.double2Str(frameDelay);
        header[3][3][1][15][1] =  width;
        header[3][3][1][15][3] =  height;
        function create(dat){
            var i,kv,data;
            data = [];
            for(i = 0; i < dat.length; i += 2){
                kv = {i : dat[i]};
                if(Array.isArray(dat[i + 1])){
                    kv.d = create(dat[i + 1]);
                }else{
                    kv.d = dat[i + 1];
                }
                data.push(kv);
            }
            return data;
        }
        return create(header);
    }
    function addCluster(){
        webmData[videoDataPos].d.push({ i: ids.Cluster,d: [{ i: ids.Timecode, d:
Math.round(clusterTimecode)}]}); // Fixed bug with Round
        clusterCounter = 0;
    }
    function addFrame(frame){
        var VP8, kfS,riff;
        riff = getWebPChunks(atob(frame.toDataURL(frameMimeType, quality).slice(23)));
```

```
        VP8 = riff.RIFF[0].WEBP[0];
        kfS = VP8.indexOf(keyframeD64Header) + 3;
        frame = {
            width: ((VP8.charCodeAt(kfS + 1) << 8) | VP8.charCodeAt(kfS)) & 0x3FFF,
            height: ((VP8.charCodeAt(kfS + 3) << 8) | VP8.charCodeAt(kfS + 2)) & 0x3FFF,
            data: VP8,
            riff: riff
        };
        if(clusterCounter > CLUSTER_MAX_DURATION){
            addCluster();
        }
        webmData[videoDataPos].d[webmData[videoDataPos].d.length-1].d.push({
            i: ids.Frame,
            d: S(ids.FrameBlock) + S( Math.round(clusterCounter) >> 8) +  S(
Math.round(clusterCounter) & 0xff) + S(128) + frame.data.slice(4),
        });
        clusterCounter += frameDelay;
        clusterTimecode += frameDelay;
        webmData[videoDataPos].d[0].d[3].d = dataTypes.double2Str(clusterTimecode);
    }
    function startEncoding(){
        frameNumber = clusterCounter = clusterTimecode = 0;
        webmData  = getHeader();
        addCluster();
    }
    function toBlob(vidData){
        var data,i,vData, len;
        vData = [];
        for(i = 0; i < vidData.length; i++){
            data = dataTypes[typeof vidData[i].d](vidData[i].d);
            len  = data.size || data.byteLength || data.length;
            vData.push(stream.num(vidData[i].i));
            vData.push(stream.compInt(len));
            vData.push(data)
        }
        return new Blob(vData, {type: videoMimeType});
    }
    function getWebPChunks(str){
        var offset, chunks, id, len, data;
        offset = 0;
        chunks = {};
        while (offset < str.length) {
            id = str.substr(offset, 4);
            // value will have top bit on (bit 32) so not simply a bitwise operation
            // Warning little endian (Will not work on big endian systems)
            len = new Uint32Array(
                new Uint8Array([
                    str.charCodeAt(offset + 7),
                    str.charCodeAt(offset + 6),
                    str.charCodeAt(offset + 5),
                    str.charCodeAt(offset + 4)
                ]).buffer)[0];
            id = str.substr(offset, 4);
            chunks[id] = chunks[id] === undefined ? [] : chunks[id];
            if (id === 'RIFF' || id === 'LIST') {
                chunks[id].push(getWebPChunks(str.substr(offset + 8, len)));
                offset += 8 + len;
            } else if (id === 'WEBP') {
                chunks[id].push(str.substr(offset + 8));
                break;
            } else {
                chunks[id].push(str.substr(offset + 4));
```

```
                break;
            }
        }
        return chunks;
    }
    function Encoder(fps, _quality = 0.8, _name = "Groover"){
        this.fps = fps;
        this.quality = quality = _quality;
        this.frameDelay = frameDelay = 1000 / fps;
        this.frame = 0;
        this.width = width = null;
        this.timecode = 0;
        this.name = name = _name;
    }
    Encoder.prototype = {
        addFrame : function(frame){
            if('canvas' in frame){
                frame = frame.canvas;
            }
            if(width === null){
                this.width = width = frame.width,
                this.height = height = frame.height
                startEncoding();
            }else
            if(width !== frame.width || height !== frame.height){
                throw RangeError("Frame size error. Frames must be the same size.");
            }
            addFrame(frame);
            this.frame += 1;
            this.timecode = clusterTimecode;
        },
        toBlob : function(){
            return toBlob(webmData);
        }
    }
    return {
        Video: Encoder,
    }
})()
```

Section 9.3: Drawing an svg image

To draw a vector SVG image, the operation is not different from a raster image :
You first need to load your SVG image into an HTMLImage element, then use the `drawImage()` method.

```
var image = new Image();
image.onload = function(){
    ctx.drawImage(this, 0,0);
}
image.src = "someFile.SVG";
```

SVG images have some advantages over raster ones, since you won't loose quality, whatever the scale you'll draw it on your canvas. But beware, it may also be a bit slower than drawing a raster image.

However, SVG images come with more restrictions than raster images.

- **For security purpose, no external content can be loaded from an SVG image referenced in an HTMLImageElement(`<img>`)**
  No external stylesheet, no external image referenced in SVGImage (**`<image/>`**) elements, no external filter or

element linked by the `xlink:href` attribute (`<use xlink:href="anImage.SVG#anElement"/>`) or the funcIRI (`url()`) attribute method etc.
Also, stylesheets appended in the main document won't have any effect on the SVG document once referenced in an HTMLImage element.
Finally, no script will be executed inside the SVG Image.
Workaround : You'll need to append all external elements inside the SVG itself before referrencing to the HTMLImage element. (for images or fonts, you need to append a dataURI version of your external resources).

- **The root element (`<svg>`) must have its width and height attributes set to an absolute value.**
If you were to use relative length (e.g `%`), then the browser won't be able to know to what it is relative. Some browsers (Blink) will try to make a guess, but most will simply ignore your image and won't draw anything, without a warning.

- **Some browsers will taint the canvas when an SVG image has been drawn to it.**
Specifically, Internet-Explorer < Edge in any case, and Safari 9 when a `<foreignObject>` is present in the SVG image.

Section 9.4: Loading and displaying an Image

To load an image and place it on the canvas

```
var image = new Image();  // see note on creating an image
image.src = "imageURL";
image.onload = function(){
    ctx.drawImage(this,0,0);
}
```

Creating an image

There are several ways to create an image

- `new Image()`
- `document.createElement("img")`
- `<img src = 'imageUrl' id='myImage'>` As part of the HTML body and retrieved with `document.getElementById('myImage')`

The image is a `HTMLImageElement`

Image.src property

The image `src`can be any valid image URL or encoded dataURL. See this topic's Remarks for more information on image formats and support.

- `image.src = "http://my.domain.com/images/myImage.jpg"`
- `image.src = "data:image/gif;base64,R0lGODlhAQABAIAAAAUEBAAAACwAAAAAAQABAAACAkQBADs="` *

*The dataURL is a 1 by 1 pixel gif image containing black

Remarks on loading and errors

The image will begin loading when its src property is set. The loading is syncriouse but the `onload` event will not be called until the function or code has exited/returned.

If you get an image from the page (for example `document.getElementById("myImage")`) and its `src` is set it may or may not have loaded. You can check on the status of the image with `HTMLImageElement.complete` which will be

`true` if complete. This does not mean the image has loaded, it means that it has either

- loaded
- there was an error
- src property has not been set and is equal to the empty String `""`

If the image is from an unreliable source and may not be accessible for a variety of reasons it will generate an error event. When this happens the image will be in a broken state. If you then attempt to draw it onto the canvas it will throw the following error

```
Uncaught DOMException: Failed to execute 'drawImage' on 'CanvasRenderingContext2D': The
HTMLImageElement provided is in the 'broken' state.
```

By supplying the `image.onerror = myImgErrorHandler` event you can take appropriate action to prevent errors.

Chapter 10: Animation

Section 10.1: Use requestAnimationFrame() NOT setInterval() for animation loops

`requestAnimationFrame` is similar to setInterval, it but has these important improvements:

- The animation code is synchronized with display refreshes for efficiency The clear + redraw code is scheduled, but not immediately executed. The browser will execute the clear + redraw code only when the display is ready to refresh. This synchronization with the refresh cycle increases your animation performance by giving your code the most available time in which to complete.
- Every loop is always completed before another loop is allowed to start. This prevents "tearing", where the user sees an incomplete version of the drawing. The eye particularly notices tearing and is distracted when tearing occurs. So preventing tearing makes your animation appear smoother and more consistent.
- Animation automatically stops when the user switches to a different browser tab. This saves power on mobile devices because the device is not wasting power computing an animation that the user can't currently see.

Device displays will refresh about 60 times per second so requestAnimationFrame can continuously redraw at about 60 "frames" per second. The eye sees motion at 20-30 frames per second so requestAnimationFrame can easily create the illusion of motion.

Notice that requestAnimationFrame is recalled at the end of each animateCircle. This is because each 'requestAnimatonFrameonly requests a single execution of the animation function.

Example: simple `requestAnimationFrame

```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    var cw=canvas.width;
    var ch=canvas.height;

    // start the animation
    requestAnimationFrame(animate);

    function animate(currentTime){

        // draw a full randomly circle
        var x=Math.random()*canvas.width;
        var y=Math.random()*canvas.height;
        var radius=10+Math.random()*15;
        ctx.beginPath();
        ctx.arc(x,y,radius,0,Math.PI*2);
        ctx.fillStyle='#'+Math.floor(Math.random()*16777215).toString(16);
```

```
        ctx.fill();

        // request another loop of animation
        requestAnimationFrame(animate);
    }

}); // end $(function(){});
</script>
</head>
<body>
    <canvas id="canvas" width=512 height=512></canvas>
</body>
</html>
```

To illustrate the advantages of requestAnimationFrame this stackoverflow question has a live demo

Section 10.2: Animate an image across the Canvas

This example loads and animates and image across the Canvas

Important Hint! Make sure you give your image time to fully load by using `image.onload`.

Annotated Code

```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    var cw=canvas.width;
    var ch=canvas.height;

    // animation related variables
    var minX=20;        // Keep the image animating
    var maxX=250;       // between minX & maxX
    var x=minX;         // The current X-coordinate
    var speedX=1;       // The image will move at 1px per loop
    var direction=1;    // The image direction: 1==righward, -1==leftward
    var y=20;           // The Y-coordinate

    // Load a new image
    // IMPORTANT!!! You must give the image time to load by using img.onload!
    var img=new Image();
    img.onload=start;
    img.src="https://dl.dropboxusercontent.com/u/139992952/stackoverflow/sun.png";
    function start(){
        // the image is fully loaded sostart animating
        requestAnimationFrame(animate);
    }

    function animate(time){
```

```
        // clear the canvas
        ctx.clearRect(0,0,cw,ch);

        // draw
        ctx.drawImage(img,x,y);

        // update
        x += speedX * direction;
        // keep "x" inside min & max
        if(x<minX){ x=minX; direction*=-1; }
        if(x>maxX){ x=maxX; direction*=-1; }

        // request another loop of animation
        requestAnimationFrame(animate);
    }

}); // end $(function(){});
</script>
</head>
<body>
    <canvas id="canvas" width=512 height=512></canvas>
</body>
</html>
```

Section 10.3: Set frame rate using requestAnimationFrame

Using requestAnimationFrame may on some systems update at more frames per second than the 60fps. 60fps is the default rate if the rendering can keep up. Some systems will run at 120fps maybe more.

If you use the following method you should only use frame rates that are integer divisions of 60 so that `(60 / FRAMES_PER_SECOND) % 1 === 0` is **`true`** or you will get inconsistent frame rates.

```
const FRAMES_PER_SECOND = 30;  // Valid values are 60,30,20,15,10...
// set the mim time to render the next frame
const FRAME_MIN_TIME = (1000/60) * (60 / FRAMES_PER_SECOND) - (1000/60) * 0.5;
var lastFrameTime = 0;  // the last frame time
function update(time){
    if(time-lastFrameTime < FRAME_MIN_TIME){ //skip the frame if the call is too early
        requestAnimationFrame(update);
        return; // return as there is nothing to do
    }
    lastFrameTime = time; // remember the time of the rendered frame
    // render the frame
    requestAnimationFrame(update); // get next farme
}
requestAnimationFrame(update); // start animation
```

Section 10.4: Easing using Robert Penners equations

An easing causes some **variable** to change **unevenly** over a **duration**.

"variable" must be able to be expressed as a number, and can represent a remarkable variety of things:

- an X-coordinate,
- a rectangle's width,
- an angle of rotation,
- the red component of an R,G,B color.
- anything that can be expressed as a number.

"duration" must be able to be expressed as a number and can also be a variety of things:

- a period of time,
- a distance to be travelled,
- a quantity of animation loops to be executed,
- anything that can be expressed as

"unevenly" means that the variable progresses from beginning to ending values unevenly:

- faster at the beginning & slower at the ending -- or visa-versa,
- overshoots the ending but backs up to the ending as the duration finishes,
- repeatedly advances/retreats elastically during the duration,
- "bounces" off the ending while coming to rest as the duration finishes.

Attribution: Robert Penner has created the "gold standard" of easing functions.

Cite: https://github.com/danro/jquery-easing/blob/master/jquery.easing.js

```
// t: elapsed time inside duration (currentTime-startTime),
// b: beginning value,
// c: total change from beginning value (endingValue-startingValue),
// d: total duration
var Easings={
    easeInQuad: function (t, b, c, d) {
      return c*(t/=d)*t + b;
    },
    easeOutQuad: function (t, b, c, d) {
      return -c *(t/=d)*(t-2) + b;
    },
    easeInOutQuad: function (t, b, c, d) {
      if ((t/=d/2) < 1) return c/2*t*t + b;
      return -c/2 * ((--t)*(t-2) - 1) + b;
    },
    easeInCubic: function (t, b, c, d) {
      return c*(t/=d)*t*t + b;
    },
    easeOutCubic: function (t, b, c, d) {
      return c*((t=t/d-1)*t*t + 1) + b;
    },
    easeInOutCubic: function (t, b, c, d) {
      if ((t/=d/2) < 1) return c/2*t*t*t + b;
      return c/2*((t-=2)*t*t + 2) + b;
    },
    easeInQuart: function (t, b, c, d) {
      return c*(t/=d)*t*t*t + b;
    },
    easeOutQuart: function (t, b, c, d) {
      return -c * ((t=t/d-1)*t*t*t - 1) + b;
    },
    easeInOutQuart: function (t, b, c, d) {
      if ((t/=d/2) < 1) return c/2*t*t*t*t + b;
      return -c/2 * ((t-=2)*t*t*t - 2) + b;
    },
    easeInQuint: function (t, b, c, d) {
      return c*(t/=d)*t*t*t*t + b;
    },
    easeOutQuint: function (t, b, c, d) {
      return c*((t=t/d-1)*t*t*t*t + 1) + b;
    },
    easeInOutQuint: function (t, b, c, d) {
```

```
    if ((t/=d/2) < 1) return c/2*t*t*t*t*t + b;
    return c/2*((t-=2)*t*t*t*t + 2) + b;
  },
  easeInSine: function (t, b, c, d) {
    return -c * Math.cos(t/d * (Math.PI/2)) + c + b;
  },
  easeOutSine: function (t, b, c, d) {
    return c * Math.sin(t/d * (Math.PI/2)) + b;
  },
  easeInOutSine: function (t, b, c, d) {
    return -c/2 * (Math.cos(Math.PI*t/d) - 1) + b;
  },
  easeInExpo: function (t, b, c, d) {
    return (t==0) ? b : c * Math.pow(2, 10 * (t/d - 1)) + b;
  },
  easeOutExpo: function (t, b, c, d) {
    return (t==d) ? b+c : c * (-Math.pow(2, -10 * t/d) + 1) + b;
  },
  easeInOutExpo: function (t, b, c, d) {
    if (t==0) return b;
    if (t==d) return b+c;
    if ((t/=d/2) < 1) return c/2 * Math.pow(2, 10 * (t - 1)) + b;
    return c/2 * (-Math.pow(2, -10 * --t) + 2) + b;
  },
  easeInCirc: function (t, b, c, d) {
    return -c * (Math.sqrt(1 - (t/=d)*t) - 1) + b;
  },
  easeOutCirc: function (t, b, c, d) {
    return c * Math.sqrt(1 - (t=t/d-1)*t) + b;
  },
  easeInOutCirc: function (t, b, c, d) {
    if ((t/=d/2) < 1) return -c/2 * (Math.sqrt(1 - t*t) - 1) + b;
    return c/2 * (Math.sqrt(1 - (t-=2)*t) + 1) + b;
  },
  easeInElastic: function (t, b, c, d) {
    var s=1.70158;var p=0;var a=c;
    if (t==0) return b;  if ((t/=d)==1) return b+c;  if (!p) p=d*.3;
    if (a < Math.abs(c)) { a=c; var s=p/4; }
    else var s = p/(2*Math.PI) * Math.asin (c/a);
    return -(a*Math.pow(2,10*(t-=1)) * Math.sin( (t*d-s)*(2*Math.PI)/p )) + b;
  },
  easeOutElastic: function (t, b, c, d) {
    var s=1.70158;var p=0;var a=c;
    if (t==0) return b;  if ((t/=d)==1) return b+c;  if (!p) p=d*.3;
    if (a < Math.abs(c)) { a=c; var s=p/4; }
    else var s = p/(2*Math.PI) * Math.asin (c/a);
    return a*Math.pow(2,-10*t) * Math.sin( (t*d-s)*(2*Math.PI)/p ) + c + b;
  },
  easeInOutElastic: function (t, b, c, d) {
    var s=1.70158;var p=0;var a=c;
    if (t==0) return b;  if ((t/=d/2)==2) return b+c;  if (!p) p=d*(.3*1.5);
    if (a < Math.abs(c)) { a=c; var s=p/4; }
    else var s = p/(2*Math.PI) * Math.asin (c/a);
    if (t < 1) return -.5*(a*Math.pow(2,10*(t-=1)) * Math.sin( (t*d-s)*(2*Math.PI)/p )) + b;
    return a*Math.pow(2,-10*(t-=1)) * Math.sin( (t*d-s)*(2*Math.PI)/p )*.5 + c + b;
  },
  easeInBack: function (t, b, c, d, s) {
    if (s == undefined) s = 1.70158;
    return c*(t/=d)*t*((s+1)*t - s) + b;
  },
  easeOutBack: function (t, b, c, d, s) {
    if (s == undefined) s = 1.70158;
```

```
    return c*((t=t/d-1)*t*((s+1)*t + s) + 1) + b;
  },
  easeInOutBack: function (t, b, c, d, s) {
    if (s == undefined) s = 1.70158;
    if ((t/=d/2) < 1) return c/2*(t*t*(((s*=(1.525))+1)*t - s)) + b;
    return c/2*((t-=2)*t*(((s*=(1.525))+1)*t + s) + 2) + b;
  },
  easeInBounce: function (t, b, c, d) {
    return c - Easings.easeOutBounce (d-t, 0, c, d) + b;
  },
  easeOutBounce: function (t, b, c, d) {
    if ((t/=d) < (1/2.75)) {
      return c*(7.5625*t*t) + b;
    } else if (t < (2/2.75)) {
      return c*(7.5625*(t-=(1.5/2.75))*t + .75) + b;
    } else if (t < (2.5/2.75)) {
      return c*(7.5625*(t-=(2.25/2.75))*t + .9375) + b;
    } else {
      return c*(7.5625*(t-=(2.625/2.75))*t + .984375) + b;
    }
  },
  easeInOutBounce: function (t, b, c, d) {
    if (t < d/2) return Easings.easeInBounce (t*2, 0, c, d) * .5 + b;
    return Easings.easeOutBounce (t*2-d, 0, c, d) * .5 + c*.5 + b;
  },
};
```

Example Usage:

```
// include the Easings object from above
var Easings = ...

// Demo
var startTime;
var beginningValue=50;  // beginning x-coordinate
var endingValue=450;    // ending x-coordinate
var totalChange=endingValue-beginningValue;
var totalDuration=3000; // ms

var keys=Object.keys(Easings);
ctx.textBaseline='middle';
requestAnimationFrame(animate);

function animate(time){
    var PI2=Math.PI*2;
    if(!startTime){startTime=time;}
    var elapsedTime=Math.min(time-startTime,totalDuration);
    ctx.clearRect(0,0,cw,ch);
    ctx.beginPath();
    for(var y=0;y<keys.length;y++){
        var key=keys[y];
        var easing=Easings[key];
        var easedX=easing(
            elapsedTime,beginningValue,totalChange,totalDuration);
        if(easedX>endingValue){easedX=endingValue;}
        ctx.moveTo(easedX,y*15);
        ctx.arc(easedX,y*15+10,5,0,PI2);
        ctx.fillText(key,460,y*15+10-1);
    }
    ctx.fill();
    if(time<startTime+totalDuration){
```

```
        requestAnimationFrame(animate);
    }
}
```

Section 10.5: Animate at a specified interval (add a new rectangle every 1 second)

This example adds a new rectangle to the canvas every 1 second (== a 1 second interval)

Annotated Code:

```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    var cw=canvas.width;
    var ch=canvas.height;

    // animation interval variables
    var nextTime=0;      // the next animation begins at "nextTime"
    var duration=1000;   // run animation every 1000ms

    var x=20;            // the X where the next rect is drawn

    // start the animation
    requestAnimationFrame(animate);

    function animate(currentTime){

        // wait for nextTime to occur
        if(currentTime<nextTime){
            // request another loop of animation
            requestAnimationFrame(animate);
            // time hasn't elapsed so just return
            return;
        }
        // set nextTime
        nextTime=currentTime+duration;

        // add another rectangle every 1000ms
        ctx.fillStyle='#'+Math.floor(Math.random()*16777215).toString(16);
        ctx.fillRect(x,30,30,30);

        // update X position for next rectangle
        x+=30;

        // request another loop of animation
        requestAnimationFrame(animate);
    }

}); // end $(function(){});
```

```
</script>
</head>
<body>
    <canvas id="canvas" width=512 height=512></canvas>
</body>
</html>
```

Section 10.6: Animate at a specified time (an animated clock)

This example animates a clock showing the seconds as a filled wedge

Annotated Code:

```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    // canvas related variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    var cw=canvas.width;
    var ch=canvas.height;
    // canvas styling for the clock
    ctx.strokeStyle='lightgray';
    ctx.fillStyle='skyblue';
    ctx.lineWidth=5;

    // cache often used values
    var PI=Math.PI;
    var fullCircle=PI*2;
    var sa=-PI/2;   // == the 12 o'clock angle in context.arc

    // start the animation
    requestAnimationFrame(animate);

    function animate(currentTime){

        // get the current seconds value from the system clock
        var date=new Date();
        var seconds=date.getSeconds();

        // clear the canvas
        ctx.clearRect(0,0,cw,ch);

        // draw a full circle (== the clock face);
        ctx.beginPath();
        ctx.moveTo(100,100);
        ctx.arc(100,100,75,0,fullCircle);
        ctx.stroke();
        // draw a wedge representing the current seconds value
        ctx.beginPath();
        ctx.moveTo(100,100);
        ctx.arc(100,100,75,sa,sa+fullCircle*seconds/60);
        ctx.fill();
```

```
        // request another loop of animation
        requestAnimationFrame(animate);
    }

}); // end $(function(){});
</script>
</head>
<body>
    <canvas id="canvas" width=512 height=512></canvas>
</body>
</html>
```

Section 10.7: Don't draw animations in your event handlers (a simple sketch app)

During `mousemove` you get flooded with 30 mouse events per second. You might not be able to redraw your drawings at 30 times per second. Even if you can, you're probably wasting computing power by drawing when the browser is not ready to draw (wasted == across display refresh cycles).

Therefore it makes sense to separate your users input events (like mousemove) from the drawing of your animations.

- In event handlers, save all the event variables that control where drawings are positioned on the Canvas. But don't actually draw anything.
- In a `requestAnimationFrame` loop, render all the drawings to the Canvas using the saved information.

By not drawing in the event handlers, you are not forcing Canvas to try to refresh complex drawings at mouse event speeds.

By doing all drawing in `requestAnimationFrame` you gain all the benefits described in here Use 'requestanimationFrame' not 'setInterval' for animation loops.

Annotated Code:

```
<!doctype html>
<html>
<head>
<style>
    body{ background-color: ivory; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    function log(){console.log.apply(console,arguments);}

    // canvas variables
    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    var cw=canvas.width;
    var ch=canvas.height;
    // set canvas styling
    ctx.strokeStyle='skyblue';
    ctx.lineJoint='round';
    ctx.lineCap='round';
    ctx.lineWidth=6;
```

```
    // handle windows scrolling & resizing
    function reOffset(){
        var BB=canvas.getBoundingClientRect();
        offsetX=BB.left;
        offsetY=BB.top;
    }
    var offsetX,offsetY;
    reOffset();
    window.onscroll=function(e){ reOffset(); }
    window.onresize=function(e){ reOffset(); }

    // vars to save points created during mousemove handling
    var points=[];
    var lastLength=0;

    // start the  animation loop
    requestAnimationFrame(draw);

    canvas.onmousemove=function(e){handleMouseMove(e);}


    function handleMouseMove(e){
        // tell the browser we're handling this event
        e.preventDefault();
        e.stopPropagation();

        // get the mouse position
        mouseX=parseInt(e.clientX-offsetX);
        mouseY=parseInt(e.clientY-offsetY);

        // save the mouse position in the points[] array
        // but don't draw anything
        points.push({x:mouseX,y:mouseY});
    }

    function draw(){
        // No additional points? Request another frame an return
        var length=points.length;
        if(length==lastLength){requestAnimationFrame(draw);return;}

        // draw the additional points
        var point=points[lastLength];
        ctx.beginPath();
        ctx.moveTo(point.x,point.y)
        for(var i=lastLength;i<length;i++){
            point=points[i];
            ctx.lineTo(point.x,point.y);
        }
        ctx.stroke();

        // request another animation loop
        requestAnimationFrame(draw);
    }

}); // end window.onload
</script>
</head>
<body>
    <h4>Move mouse over Canvas to sketch</h4>
    <canvas id="canvas" width=512 height=512></canvas>
</body>
</html>
```

Section 10.8: Simple animation with 2D context and requestAnimationFrame

This example will show you how to create a simple animation using the canvas and the 2D context. It is assumed you know how to create and add a canvas to the DOM and obtain the context

```
// this example assumes ctx and canvas have been created
const textToDisplay = "This is an example that uses the canvas to animate some text.";
const textStyle     = "white";
const BGStyle       = "black";  // background style
const textSpeed     = 0.2;      // in pixels per millisecond
const textHorMargin = 8;        // have the text a little outside the canvas


ctx.font = Math.floor(canvas.height * 0.8) + "px arial"; // size the font to 80% of canvas height
var textWidth     = ctx.measureText(textToDisplay).width; // get the text width
var totalTextSize = (canvas.width + textHorMargin * 2 + textWidth);
ctx.textBaseline  = "middle";           // not put the text in the vertical center
ctx.textAlign     = "left";             // align to the left
var textX         = canvas.width + 8;   // start with the text off screen to the right
var textOffset    = 0;                  // how far the text has moved

var startTime;
// this function is call once a frame which is approx 16.66 ms (60fps)
function update(time){                // time is passed by requestAnimationFrame
    if(startTime === undefined){    // get a reference for the start time if this is the first frame
        startTime = time;
    }
    ctx.fillStyle = BGStyle;
    ctx.fillRect(0, 0, canvas.width, canvas.height);                    // clear the canvas by
drawing over it
    textOffset    = ((time - startTime) * textSpeed) % (totalTextSize); // move the text left
    ctx.fillStyle = textStyle;                                          // set the text style
    ctx.fillText(textToDisplay, textX - textOffset, canvas.height / 2); // render the text

    requestAnimationFrame(update);// all done request the next frame
}
requestAnimationFrame(update);// to start request the first frame
```

A demo of this example at jsfiddle

Section 10.9: Animate from [x0,y0] to [x1,y1]

Use vectors to calculate incremental [x,y] from [startX,startY] to [endX,endY]

```
// dx is the total distance to move in the X direction
var dx = endX - startX;

// dy is the total distance to move in the Y direction
var dy = endY - startY;

// use a pct (percentage) to travel the total distances
// start at 0% which == the starting point
// end at 100% which == then ending point
var pct=0;

// use dx & dy to calculate where the current [x,y] is at a given pct
var x = startX + dx * pct/100;
var y = startY + dx * pct/100;
```

Example Code:

```
// canvas vars
var canvas=document.createElement("canvas");
document.body.appendChild(canvas);
canvas.style.border='1px solid red';
var ctx=canvas.getContext("2d");
var cw=canvas.width;
var ch=canvas.height;
// canvas styles
ctx.strokeStyle='skyblue';
ctx.fillStyle='blue';

// animating vars
var pct=101;
var startX=20;
var startY=50;
var endX=225;
var endY=100;
var dx=endX-startX;
var dy=endY-startY;

// start animation loop running
requestAnimationFrame(animate);

// listen for mouse events
window.onmousedown=(function(e){handleMouseDown(e);});
window.onmouseup=(function(e){handleMouseUp(e);});

// constantly running loop
// will animate dot from startX,startY to endX,endY
function animate(time){
    // demo: rerun animation
    if(++pct>100){pct=0;}
    // update
    x=startX+dx*pct/100;
    y=startY+dy*pct/100;
    // draw
    ctx.clearRect(0,0,cw,ch);
    ctx.beginPath();
    ctx.moveTo(startX,startY);
    ctx.lineTo(endX,endY);
    ctx.stroke();
    ctx.beginPath();
    ctx.arc(x,y,5,0,Math.PI*2);
    ctx.fill()
    // request another animation loop
    requestAnimationFrame(animate);
}
```

Chapter 11: Collisions and Intersections

Section 11.1: Are 2 circles colliding?

```
// circle objects: { x:, y:, radius: }
// return true if the 2 circles are colliding
// c1 and c2 are circles as defined above

function CirclesColliding(c1,c2){
    var dx=c2.x-c1.x;
    var dy=c2.y-c1.y;
    var rSum=c1.radius+c2.radius;
    return(dx*dx+dy*dy<=rSum*rSum);
}
```

Section 11.2: Are 2 rectangles colliding?

```
// rectangle objects { x:, y:, width:, height: }
// return true if the 2 rectangles are colliding
// r1 and r2 are rectangles as defined above

function RectsColliding(r1,r2){
    return !(
        r1.x>r2.x+r2.width ||
        r1.x+r1.width<r2.x ||
        r1.y>r2.y+r2.height ||
        r1.y+r1.height<r2.y
    );
}
```

Section 11.3: Are a circle and rectangle colliding?

```
// rectangle object: { x:, y:, width:, height: }
// circle object: { x:, y:, radius: }
// return true if the rectangle and circle are colliding

function RectCircleColliding(rect,circle){
    var dx=Math.abs(circle.x-(rect.x+rect.width/2));
    var dy=Math.abs(circle.y-(rect.y+rect.height/2));

    if( dx > circle.radius+rect.width/2 ){ return(false); }
    if( dy > circle.radius+rect.height/2 ){ return(false); }

    if( dx <= rect.width ){ return(true); }
    if( dy <= rect.height ){ return(true); }

    var dx=dx-rect.width;
    var dy=dy-rect.height
    return(dx*dx+dy*dy<=circle.radius*circle.radius);
}
```

Section 11.4: Are 2 line segments intercepting?

The function in this example returns `true` if two line segments are intersecting and `false` if not.

The example is designed for performance and uses closure to hold working variables

```
    // point object: {x:, y:}
    // p0 & p1 form one segment, p2 & p3 form the second segment
    // Returns true if lines segments are intercepting
    var lineSegmentsIntercept = (function(){ // function as singleton so that closure can be used

        var v1, v2, v3, cross, u1, u2;  // working variable are closed over so they do not need
creation
                                   // each time the function is called. This gives a significant
performance boost.
        v1 = {x : null, y : null}; // line p0, p1 as vector
        v2 = {x : null, y : null}; // line p2, p3 as vector
        v3 = {x : null, y : null}; // the line from p0 to p2 as vector

        function lineSegmentsIntercept (p0, p1, p2, p3) {
            v1.x = p1.x - p0.x; // line p0, p1 as vector
            v1.y = p1.y - p0.y;
            v2.x = p3.x - p2.x; // line p2, p3 as vector
            v2.y = p3.y - p2.y;
            if((cross = v1.x * v2.y - v1.y * v2.x) === 0){  // cross prod 0 if lines parallel
                return false; // no intercept
            }
            v3 = {x : p0.x - p2.x, y : p0.y - p2.y};  // the line from p0 to p2 as vector
            u2 = (v1.x * v3.y - v1.y * v3.x) / cross; // get unit distance along line p2 p3
            // code point B
            if (u2 >= 0 && u2 <= 1){                   // is intercept on line p2, p3
                u1 = (v2.x * v3.y - v2.y * v3.x) / cross; // get unit distance on line p0, p1;
                // code point A
                return (u1 >= 0 && u1 <= 1);           // return true if on line else false.
                // code point A end
            }
            return false; // no intercept;
            // code point B end
        }
        return lineSegmentsIntercept;  // return function with closure for optimisation.
    })();
```

Usage example

```
var p1 = {x: 100, y: 0};   // line 1
var p2 = {x: 120, y: 200};
var p3 = {x: 0,   y: 100}; // line 2
var p4 = {x: 100, y: 120};
var areIntersepting = lineSegmentsIntercept (p1, p2, p3, p4); // true
```

The example is easily modified to return the point of intercept. Replace the code between `code point A` and `A end` with

```
if(u1 >= 0 && u1 <= 1){
    return {
        x : p0.x + v1.x * u1,
        y : p0.y + v1.y * u1,
    };
}
```

Or if you want to get the intercept point on the lines, ignoring the line segments start and ends replace the code between `code point B` and `B end` with

```
return {
    x : p2.x + v2.x * u2,
```

```
    y : p2.y + v2.y * u2,
};
```

Both modifications will return false if there is no intercept or return the point of intercept as `{x : xCoord, y : yCoord}`

Section 11.5: Are a line segment and circle colliding?

```
// [x0,y0] to [x1,y1] define a line segment
// [cx,cy] is circle centerpoint, cr is circle radius
function isCircleSegmentColliding(x0,y0,x1,y1,cx,cy,cr){

    // calc delta distance: source point to line start
    var dx=cx-x0;
    var dy=cy-y0;

    // calc delta distance: line start to end
    var dxx=x1-x0;
    var dyy=y1-y0;

    // Calc position on line normalized between 0.00 & 1.00
    // == dot product divided by delta line distances squared
    var t=(dx*dxx+dy*dyy)/(dxx*dxx+dyy*dyy);

    // calc nearest pt on line
    var x=x0+dxx*t;
    var y=y0+dyy*t;

    // clamp results to being on the segment
    if(t<0){x=x0;y=y0;}
    if(t>1){x=x1;y=y1;}

    return( (cx-x)*(cx-x)+(cy-y)*(cy-y) < cr*cr );
}
```

Section 11.6: Are line segment and rectangle colliding?

```
// var rect={x:,y:,width:,height:};
// var line={x1:,y1:,x2:,y2:};
// Get interseting point of line segment & rectangle (if any)
function lineRectCollide(line,rect){

    // p=line startpoint, p2=line endpoint
    var p={x:line.x1,y:line.y1};
    var p2={x:line.x2,y:line.y2};

    // top rect line
    var q={x:rect.x,y:rect.y};
    var q2={x:rect.x+rect.width,y:rect.y};
    if(lineSegmentsCollide(p,p2,q,q2)){ return true; }
    // right rect line
    var q=q2;
    var q2={x:rect.x+rect.width,y:rect.y+rect.height};
    if(lineSegmentsCollide(p,p2,q,q2)){ return true; }
    // bottom rect line
    var q=q2;
    var q2={x:rect.x,y:rect.y+rect.height};
    if(lineSegmentsCollide(p,p2,q,q2)){ return true; }
    // left rect line
    var q=q2;
```

```
    var q2={x:rect.x,y:rect.y};
    if(lineSegmentsCollide(p,p2,q,q2)){ return true; }

    // not intersecting with any of the 4 rect sides
    return(false);
}

// point object: {x:, y:}
// p0 & p1 form one segment, p2 & p3 form the second segment
// Get interseting point of 2 line segments (if any)
// Attribution: http://paulbourke.net/geometry/pointlineplane/
function lineSegmentsCollide(p0,p1,p2,p3) {

    var unknownA = (p3.x-p2.x) * (p0.y-p2.y) - (p3.y-p2.y) * (p0.x-p2.x);
    var unknownB = (p1.x-p0.x) * (p0.y-p2.y) - (p1.y-p0.y) * (p0.x-p2.x);
    var denominator  = (p3.y-p2.y) * (p1.x-p0.x) - (p3.x-p2.x) * (p1.y-p0.y);

    // Test if Coincident
    // If the denominator and numerator for the ua and ub are 0
    //    then the two lines are coincident.
    if(unknownA==0 && unknownB==0 && denominator==0){return(null);}

    // Test if Parallel
    // If the denominator for the equations for ua and ub is 0
    //     then the two lines are parallel.
    if (denominator == 0) return null;

    // test if line segments are colliding
    unknownA /= denominator;
    unknownB /= denominator;
    var isIntersecting=(unknownA>=0 && unknownA<=1 && unknownB>=0 && unknownB<=1)

    return(isIntersecting);
}
```

Section 11.7: Are 2 convex polygons colliding?

Use the Separating Axis Theorem to determine if 2 convex polygons are intersecting

THE POLYGONS MUST BE CONVEX

Attribution: Markus Jarderot @ How to check intersection between 2 rotated rectangles?

```
// polygon objects are an array of vertices forming the polygon
//     var polygon1=[{x:100,y:100},{x:150,y:150},{x:50,y:150},...];
// THE POLYGONS MUST BE CONVEX
// return true if the 2 polygons are colliding

function convexPolygonsCollide(a, b){
    var polygons = [a, b];
    var minA, maxA, projected, i, i1, j, minB, maxB;

    for (i = 0; i < polygons.length; i++) {

        // for each polygon, look at each edge of the polygon, and determine if it separates
        // the two shapes
        var polygon = polygons[i];
        for (i1 = 0; i1 < polygon.length; i1++) {

            // grab 2 vertices to create an edge
            var i2 = (i1 + 1) % polygon.length;
```

```
            var p1 = polygon[i1];
            var p2 = polygon[i2];

            // find the line perpendicular to this edge
            var normal = { x: p2.y - p1.y, y: p1.x - p2.x };

            minA = maxA = undefined;
            // for each vertex in the first shape, project it onto the line perpendicular to the
edge
            // and keep track of the min and max of these values
            for (j = 0; j < a.length; j++) {
                projected = normal.x * a[j].x + normal.y * a[j].y;
                if (minA==undefined || projected < minA) {
                    minA = projected;
                }
                if (maxA==undefined || projected > maxA) {
                    maxA = projected;
                }
            }

            // for each vertex in the second shape, project it onto the line perpendicular to the
edge
            // and keep track of the min and max of these values
            minB = maxB = undefined;
            for (j = 0; j < b.length; j++) {
                projected = normal.x * b[j].x + normal.y * b[j].y;
                if (minB==undefined || projected < minB) {
                    minB = projected;
                }
                if (maxB==undefined || projected > maxB) {
                    maxB = projected;
                }
            }

            // if there is no overlap between the projects, the edge we are looking at separates the
two
            // polygons, and we know there is no overlap
            if (maxA < minB || maxB < minA) {
                return false;
            }
        }
    }
    return true;
};
```

Section 11.8: Are 2 polygons colliding? (both concave and convex polys are allowed)

Tests all polygon sides for intersections to determine if 2 polygons are colliding.

```
// polygon objects are an array of vertices forming the polygon
//     var polygon1=[{x:100,y:100},{x:150,y:150},{x:50,y:150},...];
// The polygons can be both concave and convex
// return true if the 2 polygons are colliding

function polygonsCollide(p1,p2){
    // turn vertices into line points
    var lines1=verticesToLinePoints(p1);
    var lines2=verticesToLinePoints(p2);
    // test each poly1 side vs each poly2 side for intersections
    for(i=0; i<lines1.length; i++){
```

```
    for(j=0; j<lines2.length; j++){
        // test if sides intersect
        var p0=lines1[i][0];
        var p1=lines1[i][1];
        var p2=lines2[j][0];
        var p3=lines2[j][1];
        // found an intersection -- polys do collide
        if(lineSegmentsCollide(p0,p1,p2,p3)){return(true);}
    }}
    // none of the sides intersect
    return(false);
}
// helper: turn vertices into line points
function verticesToLinePoints(p){
    // make sure polys are self-closing
    if(!(p[0].x==p[p.length-1].x && p[0].y==p[p.length-1].y)){
        p.push({x:p[0].x,y:p[0].y});
    }
    var lines=[];
    for(var i=1;i<p.length;i++){
        var p1=p[i-1];
        var p2=p[i];
        lines.push([
            {x:p1.x, y:p1.y},
            {x:p2.x, y:p2.y}
        ]);
    }
    return(lines);
}
// helper: test line intersections
// point object: {x:, y:}
// p0 & p1 form one segment, p2 & p3 form the second segment
// Get interseting point of 2 line segments (if any)
// Attribution: http://paulbourke.net/geometry/pointlineplane/
function lineSegmentsCollide(p0,p1,p2,p3) {
    var unknownA = (p3.x-p2.x) * (p0.y-p2.y) - (p3.y-p2.y) * (p0.x-p2.x);
    var unknownB = (p1.x-p0.x) * (p0.y-p2.y) - (p1.y-p0.y) * (p0.x-p2.x);
    var denominator  = (p3.y-p2.y) * (p1.x-p0.x) - (p3.x-p2.x) * (p1.y-p0.y);

    // Test if Coincident
    // If the denominator and numerator for the ua and ub are 0
    //    then the two lines are coincident.
    if(unknownA==0 && unknownB==0 && denominator==0){return(null);}

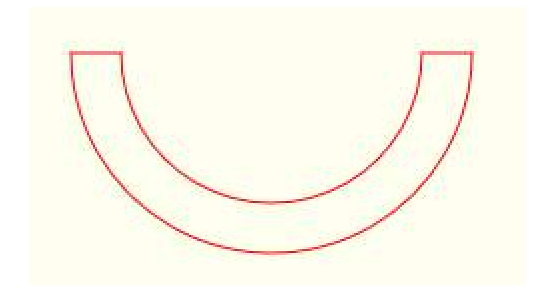
    // Test if Parallel
    // If the denominator for the equations for ua and ub is 0
    //     then the two lines are parallel.
    if (denominator == 0) return null;

    // test if line segments are colliding
    unknownA /= denominator;
    unknownB /= denominator;
    var isIntersecting=(unknownA>=0 && unknownA<=1 && unknownB>=0 && unknownB<=1)

    return(isIntersecting);
}
```

Section 11.9: Is an X,Y point inside an arc?

Tests if the [x,y] point is inside a closed arc.

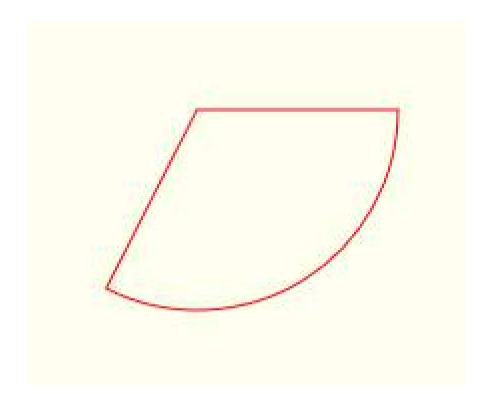


```
var arc={
    cx:150, cy:150,
    innerRadius:75, outerRadius:100,
    startAngle:0, endAngle:Math.PI
}

function isPointInArc(x,y,arc){
    var dx=x-arc.cx;
    var dy=y-arc.cy;
    var dxy=dx*dx+dy*dy;
    var rrOuter=arc.outerRadius*arc.outerRadius;
    var rrInner=arc.innerRadius*arc.innerRadius;
    if(dxy<rrInner || dxy>rrOuter){return(false);}
    var angle=(Math.atan2(dy,dx)+PI2)%PI2;
    return(angle>=arc.startAngle && angle<=arc.endAngle);
}
```

Section 11.10: Is an X,Y point inside a wedge?

Tests if the [x,y] point is inside a wedge.



```
// wedge objects: {cx:,cy:,radius:,startAngle:,endAngle:}
// var wedge={
//     cx:150, cy:150,  // centerpoint
//     radius:100,
//     startAngle:0, endAngle:Math.PI
// }
// Return true if the x,y point is inside the closed wedge

function isPointInWedge(x,y,wedge){
    var PI2=Math.PI*2;
    var dx=x-wedge.cx;
    var dy=y-wedge.cy;
    var rr=wedge.radius*wedge.radius;
    if(dx*dx+dy*dy>rr){return(false);}
    var angle=(Math.atan2(dy,dx)+PI2)%PI2;
    return(angle>=wedge.startAngle && angle<=wedge.endAngle);
}
```

Section 11.11: Is an X,Y point inside a circle?

Tests if an [x,y] point is inside a circle.

```
// circle objects: {cx:,cy:,radius:,startAngle:,endAngle:}
// var circle={
//     cx:150, cy:150,  // centerpoint
//     radius:100,
// }
// Return true if the x,y point is inside the circle

function isPointInCircle(x,y,circle){
    var dx=x-circle.cx;
    var dy=y-circle.cy;
    return(dx*dx+dy*dy<circle.radius*circle.radius);
}
```

Section 11.12: Is an X,Y point inside a rectangle?

Tests if an [x,y] point is inside a rectangle.

```
// rectangle objects: {x:, y:, width:, height: }
// var rect={x:10, y:15, width:25, height:20}
// Return true if the x,y point is inside the rectangle

function isPointInRectangle(x,y,rect){
    return(x>rect.x && x<rect.x+rect.width && y>rect.y && y<rect.y+rect.height);
}
```

Chapter 12: Clearing the screen

Section 12.1: Rectangles

You can use the `clearRect` method to clear any rectangular section of the canvas.

```
// Clear the entire canvas
ctx.clearRect(0, 0, canvas.width, canvas.height);
```

> **Note:** `clearRect` is dependent on the transformation matrix.

To deal with this, it's possible to reset the transformation matrix before you clear the canvas.

```
ctx.save();                                        // Save the current context state
ctx.setTransform(1, 0, 0, 1, 0, 0);                // Reset the transformation matrix
ctx.clearRect(0, 0, canvas.width, canvas.height); // Clear the canvas
ctx.restore();                                     // Revert context state including
                                                   // transformation matrix
```

> **Note:** `ctx.save` and `ctx.restore` are only requiered if you wish to keep the canvas 2D context state. In some situations save and restore can be be slow and generally should be avoided if not required.

Section 12.2: Clear canvas with gradient

Rather than use `clearRect` which makes all pixels transparent you may want a background.

To clear with a gradient

```
// create the background gradient once
var bgGrad = ctx.createLinearGradient(0,0,0,canvas.height);
bgGrad.addColorStop(0,"#0FF");
bgGrad.addColorStop(1,"#08F");

// Every time you need to clear the canvas
ctx.fillStyle = bgGrad;
ctx.fillRect(0,0,canvas.width,canvas.height);
```

This is about half as quick `0.008ms` as clearRect `0.004ms` but the 4millions of a second should not negatively impact any realtime animation. (Times will vary considerably depending on device, resolution, browser, and browser configuration. Times are for comparison only)

Section 12.3: Clear canvas using composite operation

Clear the canvas using compositing operation. This will clear the canvas independent of transforms but is not as fast as `clearRect()`.

```
ctx.globalCompositeOperation = 'copy';
```

anything drawn next will clear previous content.

Section 12.4: Raw image data

It's possible to write directly to the rendered image data using `putImageData`. By creating new image data then assigning it to the canvas, you will clear the entire screen.

```
var imageData = ctx.createImageData(canvas.width, canvas.height);
ctx.putImageData(imageData, 0, 0);
```

Note: `putImageData` *is not affected by any transformations applied to the context. It will write data directly to the rendered pixel region.*

Section 12.5: Complex shapes

It's possible to clear complex shaped regions by changing the `globalCompositeOperation` property.

```
// All pixels being drawn will be transparent
ctx.globalCompositeOperation = 'destination-out';

// Clear a triangular section
ctx.globalAlpha = 1;    // ensure alpha is 1
ctx.fillStyle = '#000'; // ensure the current fillStyle does not have any transparency
ctx.beginPath();
ctx.moveTo(10, 0);
ctx.lineTo(0, 10);
ctx.lineTo(20, 10);
ctx.fill();

// Begin drawing normally again
ctx.globalCompositeOperation = 'source-over';
```

Chapter 13: Responsive Design

Section 13.1: Creating a responsive full page canvas

Starter code to create and remove a full page canvas that responds to resize events via JavaScript.

```
var canvas;    // Global canvas reference
var ctx;       // Global 2D context reference
// Creates a canvas
function createCanvas () {
    const canvas = document.createElement("canvas");
    canvas.style.position = "absolute"; // Set the style
    canvas.style.left     = "0px";      // Position in top left
    canvas.style.top      = "0px";
    canvas.style.zIndex   = 1;
    document.body.appendChild(canvas);  // Add to document
    return canvas;
}
// Resizes canvas. Will create a canvas if it does not exist
function sizeCanvas () {
    if (canvas === undefined) {         // Check for global canvas reference
        canvas = createCanvas();        // Create a new canvas element
        ctx = canvas.getContext("2d");  // Get the 2D context
    }
    canvas.width  = innerWidth;         // Set the canvas resolution to fill the page
    canvas.height = innerHeight;
}
// Removes the canvas
function removeCanvas () {
    if (canvas !== undefined) {              // Make sure there is something to remove
        removeEventListener("resize", sizeCanvas); // Remove resize event
        document.body.removeChild(canvas);   // Remove the canvas from the DOM
        ctx = undefined;                     // Dereference the context
        canvas = undefined;                  // Dereference the canvas
     }
}

// Add the resize listener
addEventListener("resize", sizeCanvas);
// Call sizeCanvas to create and set the canvas resolution
sizeCanvas();
// ctx and canvas are now available for use.
```

If you no longer need the canvas you can remove it by calling `removeCanvas()`

A demo of this example at jsfiddle

Section 13.2: Mouse coordinates after resizing (or scrolling)

Canvas apps often rely heavily on user interaction with the mouse, but when the window is resized, the mouse event coordinates that canvas relies on are likely changed because resizing causes the canvas to be offset in a different position relative to the window. Thus, responsive design requires that the canvas offset position be recalculated when the window is resized -- and also recalculated when the window is scrolled.

This code listens for window resizing events and recalculates the offsets used in mouse event handlers:

```
// variables holding the current canvas offset position
//    relative to the window
```

```
var offsetX,offsetY;

// a function to recalculate the canvas offsets
function reOffset(){
    var BB=canvas.getBoundingClientRect();
    offsetX=BB.left;
    offsetY=BB.top;
}

// listen for window resizing (and scrolling) events
//     and then recalculate the canvas offsets
window.onscroll=function(e){ reOffset(); }
window.onresize=function(e){ reOffset(); }

// example usage of the offsets in a mouse handler
function handleMouseUp(e){
    // use offsetX & offsetY to get the correct mouse position
    mouseX=parseInt(e.clientX-offsetX);
    mouseY=parseInt(e.clientY-offsetY);
    // ...
}
```

Section 13.3: Responsive canvas animations without resize events

The window resize events can fire in response to the movement of the user's input device. When you resize a canvas it is automatically cleared and you are forced to re-render the content. For animations you do this every frame via the main loop function called by `requestAnimationFrame` which does its best to keep the rendering in sync with the display hardware.

The problem with the resize event is that when the mouse is used to resize the window the events can be trigger many times quicker than the standard 60fps rate of the browser. When the resize event exits the canvas back buffer is presented to the DOM out of sync with the display device, which can cause shearing and other negative effects. There is also a lot of needless memory allocation and release that can further impact the animation when GC cleans up some time afterwards.

Debounced resize event

A common way to deal with the high firing rates of the resize event is to debounce the resize event.

```
// Assume canvas is in scope
addEventListener.("resize", debouncedResize );

// debounce timeout handle
var debounceTimeoutHandle;

// The debounce time in ms (1/1000th second)
const DEBOUNCE_TIME = 100;

// Resize function
function debouncedResize () {
    clearTimeout(debounceTimeoutHandle);  // Clears any pending debounce events

    // Schedule a canvas resize
    debounceTimeoutHandle = setTimeout(resizeCanvas, DEBOUNCE_TIME);
}
```

```
// canvas resize function
function resizeCanvas () { ... resize and redraw ... }
```

The above example delays the resizing of the canvas until 100ms after the resize event. If in that time further resize events are triggered the existing resize timeout is canceled and a new one scheduled. This effectively consumes most of the resize events.

It still has some problems, the most notable is the delay between resizing and seeing the resized canvas. Reducing the debounce time improves this but the resize is still out of sync with the display device. You also still have the animation main loop rendering to an ill fitting canvas.

More code can reduce the problems! More code also creates its own new problems.

Simple and the best resize

Having tried many differing ways to smooth out the resizing of the canvas, from the absurdly complex, to just ignoring the problem (who cares anyways?) I fell back to a trusty friend.

K.I.S.S. is something most programmers should be aware of ((**K**eep **I**t **S**imple **S**tupid) *The stupid refers to me for not having thought of it years ago.*) and it turns out the best solution is the simplest of all.

Just resize the canvas from within the main animation loop. It stays in sync with the display device, there is no needless rendering, and the resource management is at the minimum possible while maintaining full frame rate. Nor do you need to add a resize event to the window or any additional resize functions.

You add the resize where you would normally clear the canvas by checking if the canvas size matches the window size. If not resize it.

```
// Assumes canvas element is in scope as canvas

// Standard main loop function callback from requestAnimationFrame
function mainLoop(time) {

    // Check if the canvas size matches the window size
    if (canvas.width !== innerWidth || canvas.height !== innerHeight) {
        canvas.width = innerWidth;    // resize canvas
        canvas.height = innerHeight;  // also clears the canvas
    } else {
        ctx.clearRect(0, 0, canvas.width, canvas.height); // clear if not resized
    }

    // Animation code as normal.



    requestAnimationFrame(mainLoop);
}
```

Chapter 14: Shadows

Section 14.1: Sticker effect using shadows

This code adds outwardly increasing shadows to an image to create a "sticker" version of the image.

Notes:

- In addition to being an ImageObject, the "img" argument can also be a Canvas element. This allows you to stickerize your own custom drawings. If you draw text on the Canvas argument, you can also stickerize that text.
- Fully opaque images will have no sticker effect because the effect is drawn around clusters of opaque pixels that are bordered by transparent pixels.



```
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
document.body.appendChild(canvas);
canvas.style.background='navy';
canvas.style.border='1px solid red;';

// Always(!) wait for your images to fully load before trying to drawImage them!
var img=new Image();
img.onload=start;
// put your img.src here...
img.src='http://i.stack.imgur.com/bXaB6.png';
function start(){
    ctx.drawImage(img,20,20);
    var sticker=stickerEffect(img,5);
    ctx.drawImage(sticker, 150,20);
}

function stickerEffect(img,grow){
    var canvas1=document.createElement("canvas");
    var ctx1=canvas1.getContext("2d");
    var canvas2=document.createElement("canvas");
    var ctx2=canvas2.getContext("2d");
    canvas1.width=canvas2.width=img.width+grow*2;
    canvas1.height=canvas2.height=img.height+grow*2;
    ctx1.drawImage(img,grow,grow);
    ctx2.shadowColor='white';
    ctx2.shadowBlur=2;
    for(var i=0;i<grow;i++){
        ctx2.drawImage(canvas1,0,0);
        ctx1.drawImage(canvas2,0,0);
    }
    ctx2.shadowColor='rgba(0,0,0,0)';
    ctx2.drawImage(img,grow,grow);
    return(canvas2);
```

```
}
```

Section 14.2: How to stop further shadowing

Once shadowing is turned on, every new drawing to the canvas will be shadowed.

Turn off further shadowing by setting `context.shadowColor` to a transparent color.

```
// start shadowing
context.shadowColor='black';

... render some shadowed drawings ...

// turn off shadowing.
context.shadowColor='rgba(0,0,0,0)';
```

Section 14.3: Shadowing is computationally expensive -- Cache that shadow!

Warning! Apply shadows sparingly!

Applying shadowing is expensive and is multiplicatively expensive if you apply shadowing inside an animation loop.

Instead, cache a shadowed version of your image (or other drawing):

- At the start of your app, create a shadowed version of your image in a second in-memory-only Canvas: `var memoryCanvas = document.createElement('canvas') ...`
- Whenever you need the shadowed version, draw that pre-shadowed image from the in-memory canvas to the visible canvas: `context.drawImage(memoryCanvas,x,y)`



```
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
var cw=canvas.width;
var ch=canvas.height;
canvas.style.border='1px solid red;';
document.body.appendChild(canvas);

// Always(!) use "img.onload" to give your image time to
//     fully load before you try drawing it to the Canvas!
var img=new Image();
img.onload=start;
// Put your own img.src here
img.src="http://i.stack.imgur.com/hYFNe.png";
function start(){
    ctx.drawImage(img,0,20);
    var cached=cacheShadowedImage(img,'black',5,3,3);
    for(var i=0;i<5;i++){
```

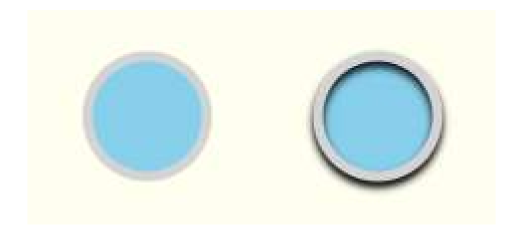
```
        ctx.drawImage(cached,i*(img.width+10),80);
    }
}

function cacheShadowedImage(img,shadowcolor,blur){
    var c=document.createElement('canvas');
    var cctx=c.getContext('2d');
    c.width=img.width+blur*2+2;
    c.height=img.height+blur*2+2;
    cctx.shadowColor=shadowcolor;
    cctx.shadowBlur=blur;
    cctx.drawImage(img,blur+1,blur+1);
    return(c);
}
```

Section 14.4: Add visual depth with shadows

The traditional use of shadowing is to give 2-dimensional drawings the illusion of 3D depth.

This example shows the same "button" with and without shadowing



```
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
document.body.appendChild(canvas);

ctx.fillStyle='skyblue';
ctx.strokeStyle='lightgray';
ctx.lineWidth=5;

// without shadow
ctx.beginPath();
ctx.arc(60,60,30,0,Math.PI*2);
ctx.closePath();
ctx.fill();
ctx.stroke();

// with shadow
ctx.shadowColor='black';
ctx.shadowBlur=4;
ctx.shadowOffsetY=3;
ctx.beginPath();
ctx.arc(175,60,30,0,Math.PI*2);
ctx.closePath();
ctx.fill();
ctx.stroke();
// stop the shadowing
ctx.shadowColor='rgba(0,0,0,0)';
```

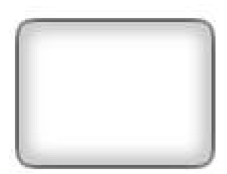
Section 14.5: Inner shadows

Canvas does not have CSS's `inner-shadow`.

- Canvas will shadow the outside of a filled shape.
- Canvas will shadow both inside and outside a stroked shape.

But it's easy to create inner-shadows using compositing.

Strokes with an inner-shadow



To create strokes with an inner-shadow, use `destination-in` compositing which causes existing content to remain only where existing content is overlapped by new content. Existing content that is not overlapped by new content is erased.

1. **Stroke a shape with a shadow.** The shadow will extend both outward and inward from the stroke. We must get rid of the outer-shadow -- leaving just the desired inner-shadow.
2. **Set compositing to `destination-in`** which keeps the existing stroked shadow only where it is overlapped by any new drawings.
3. **Fill the shape.** This causes the stroke and inner-shadow to remain while the outer shadow is erased. *Well, not exactly! Since a stroke is half-inside and half-outside the filled shape, the outside half of the stroke will be erased also. The fix is to double the `context.lineWidth` so half of the double-sized stroke is still inside the filled shape.*

```
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
document.body.appendChild(canvas);

// draw an opaque shape -- here we use a rounded rectangle
defineRoundedRect(30,30,100,75,10);

// set shadowing
ctx.shadowColor='black';
ctx.shadowBlur=10;

// stroke the shadowed rounded rectangle
ctx.lineWidth=4;
ctx.stroke();

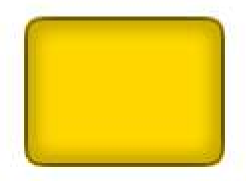
// set compositing to erase everything outside the stroke
ctx.globalCompositeOperation='destination-in';
ctx.fill();

// always clean up -- set compsiting back to default
ctx.globalCompositeOperation='source-over';


function defineRoundedRect(x,y,width,height,radius) {
    ctx.beginPath();
    ctx.moveTo(x + radius, y);
    ctx.lineTo(x + width - radius, y);
    ctx.quadraticCurveTo(x + width, y, x + width, y + radius);
    ctx.lineTo(x + width, y + height - radius);
    ctx.quadraticCurveTo(x + width, y + height, x + width - radius, y + height);
    ctx.lineTo(x + radius, y + height);
    ctx.quadraticCurveTo(x, y + height, x, y + height - radius);
```

```
    ctx.lineTo(x, y + radius);
    ctx.quadraticCurveTo(x, y, x + radius, y);
    ctx.closePath();
}
```

Stroked Fills with an inner-shadow



To create fills with an inner-shadow, follow steps #1-3 above but further use `destination-over` compositing which causes new content to be drawn **under existing content**.

4. **Set compositing to `destination-over`** which causes the fill to be drawn **under** the existing inner-shadow.
5. **Turn off shadowing** by setting `context.shadowColor` to a transparent color.
6. **Fill the shape** with the desired color. The shape will be filled underneath the existing inner-shadow.

```
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
document.body.appendChild(canvas);

// draw an opaque shape -- here we use a rounded rectangle
defineRoundedRect(30,30,100,75,10);

// set shadowing
ctx.shadowColor='black';
ctx.shadowBlur=10;

// stroke the shadowed rounded rectangle
ctx.lineWidth=4;
ctx.stroke();

// stop shadowing
ctx.shadowColor='rgba(0,0,0,0)';

// set compositing to erase everything outside the stroke
ctx.globalCompositeOperation='destination-in';
ctx.fill();

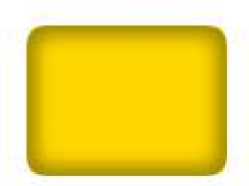
// set compositing to erase everything outside the stroke
ctx.globalCompositeOperation='destination-over';
ctx.fillStyle='gold';
ctx.fill();

// always clean up -- set compsiting back to default
ctx.globalCompositeOperation='source-over';

function defineRoundedRect(x,y,width,height,radius) {
    ctx.beginPath();
    ctx.moveTo(x + radius, y);
    ctx.lineTo(x + width - radius, y);
    ctx.quadraticCurveTo(x + width, y, x + width, y + radius);
    ctx.lineTo(x + width, y + height - radius);
    ctx.quadraticCurveTo(x + width, y + height, x + width - radius, y + height);
    ctx.lineTo(x + radius, y + height);
    ctx.quadraticCurveTo(x, y + height, x, y + height - radius);
    ctx.lineTo(x, y + radius);
    ctx.quadraticCurveTo(x, y, x + radius, y);
```

```
    ctx.closePath();
}
```

Non-stroked Fills with an inner-shadow



To draw a filled shape with an inner-shadow, but with no stroke, you can draw the stroke off-canvas and use `shadowOffsetX` to push the shadow back onto the canvas.

```
var canvas=document.createElement("canvas");
var ctx=canvas.getContext("2d");
document.body.appendChild(canvas);

// define an opaque shape -- here we use a rounded rectangle
defineRoundedRect(30-500,30,100,75,10);

// set shadowing
ctx.shadowColor='black';
ctx.shadowBlur=10;
ctx.shadowOffsetX=500;

// stroke the shadowed rounded rectangle
ctx.lineWidth=4;
ctx.stroke();

// stop shadowing
ctx.shadowColor='rgba(0,0,0,0)';

// redefine an opaque shape -- here we use a rounded rectangle
defineRoundedRect(30,30,100,75,10);

// set compositing to erase everything outside the stroke
ctx.globalCompositeOperation='destination-in';
ctx.fill();

// set compositing to erase everything outside the stroke
ctx.globalCompositeOperation='destination-over';
ctx.fillStyle='gold';
ctx.fill();

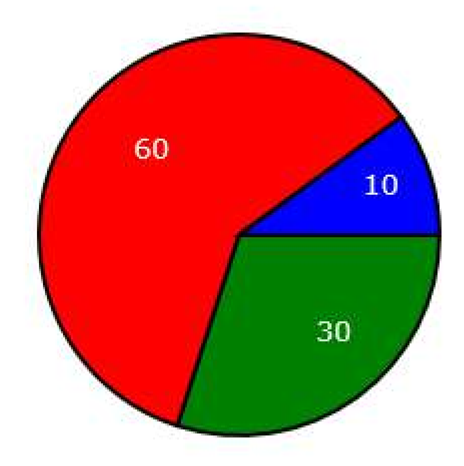
// always clean up -- set compsiting back to default
ctx.globalCompositeOperation='source-over';

function defineRoundedRect(x,y,width,height,radius) {
    ctx.beginPath();
    ctx.moveTo(x + radius, y);
    ctx.lineTo(x + width - radius, y);
    ctx.quadraticCurveTo(x + width, y, x + width, y + radius);
    ctx.lineTo(x + width, y + height - radius);
    ctx.quadraticCurveTo(x + width, y + height, x + width - radius, y + height);
    ctx.lineTo(x + radius, y + height);
    ctx.quadraticCurveTo(x, y + height, x, y + height - radius);
    ctx.lineTo(x, y + radius);
    ctx.quadraticCurveTo(x, y, x + radius, y);
    ctx.closePath();
```

```
}
```

Chapter 15: Charts & Diagrams

Section 15.1: Pie Chart with Demo




```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    var canvas = document.getElementById("canvas");
    var ctx = canvas.getContext("2d");
    ctx.lineWidth = 2;
    ctx.font = '14px verdana';

    var PI2 = Math.PI * 2;
    var myColor = ["Green", "Red", "Blue"];
    var myData = [30, 60, 10];
    var cx = 150;
    var cy = 150;
    var radius = 100;

    pieChart(myData, myColor);

    function pieChart(data, colors) {
      var total = 0;
      for (var i = 0; i < data.length; i++) {
        total += data[i];
      }

      var sweeps = []
      for (var i = 0; i < data.length; i++) {
        sweeps.push(data[i] / total * PI2);
      }

      var accumAngle = 0;
      for (var i = 0; i < sweeps.length; i++) {
        drawWedge(accumAngle, accumAngle + sweeps[i], colors[i], data[i]);
        accumAngle += sweeps[i];
      }
```

```
    }

    function drawWedge(startAngle, endAngle, fill, label) {
      // draw the wedge
      ctx.beginPath();
      ctx.moveTo(cx, cy);
      ctx.arc(cx, cy, radius, startAngle, endAngle, false);
      ctx.closePath();
      ctx.fillStyle = fill;
      ctx.strokeStyle = 'black';
      ctx.fill();
      ctx.stroke();

      // draw the label
      var midAngle = startAngle + (endAngle - startAngle) / 2;
      var labelRadius = radius * .65;
      var x = cx + (labelRadius) * Math.cos(midAngle);
      var y = cy + (labelRadius) * Math.sin(midAngle);
      ctx.fillStyle = 'white';
      ctx.fillText(label, x, y);
    }

}); // end $(function(){});
</script>
</head>
<body>
    <canvas id="canvas" width=512 height=512></canvas>
</body>
</html>
```

Section 15.2: Line with arrowheads



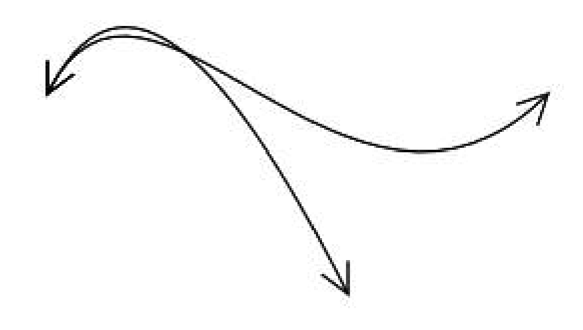
```
// Usage:
drawLineWithArrows(50,50,150,50,5,8,true,true);

// x0,y0: the line's starting point
// x1,y1: the line's ending point
// width: the distance the arrowhead perpendicularly extends away from the line
// height: the distance the arrowhead extends backward from the endpoint
// arrowStart: true/false directing to draw arrowhead at the line's starting point
// arrowEnd: true/false directing to draw arrowhead at the line's ending point

function drawLineWithArrows(x0,y0,x1,y1,aWidth,aLength,arrowStart,arrowEnd){
    var dx=x1-x0;
    var dy=y1-y0;
    var angle=Math.atan2(dy,dx);
    var length=Math.sqrt(dx*dx+dy*dy);
    //
    ctx.translate(x0,y0);
    ctx.rotate(angle);
    ctx.beginPath();
    ctx.moveTo(0,0);
    ctx.lineTo(length,0);
```

```
    if(arrowStart){
        ctx.moveTo(aLength,-aWidth);
        ctx.lineTo(0,0);
        ctx.lineTo(aLength,aWidth);
    }
    if(arrowEnd){
        ctx.moveTo(length-aLength,-aWidth);
        ctx.lineTo(length,0);
        ctx.lineTo(length-aLength,aWidth);
    }
    //
    ctx.stroke();
    ctx.setTransform(1,0,0,1,0,0);
}
```

Section 15.3: Cubic & Quadratic Bezier curve with arrowheads



```
// Usage:
var p0={x:50,y:100};
var p1={x:100,y:0};
var p2={x:200,y:200};
var p3={x:300,y:100};

cubicCurveArrowHeads(p0, p1, p2, p3, 15, true, true);

quadraticCurveArrowHeads(p0, p1, p2, 15, true, true);

// or use defaults true for both ends with arrow heads
cubicCurveArrowHeads(p0, p1, p2, p3, 15);

quadraticCurveArrowHeads(p0, p1, p2, 15);



// draws both cubic and quadratic bezier
function bezWithArrowheads(p0, p1, p2, p3, arrowLength, hasStartArrow, hasEndArrow) {
    var x, y, norm, ex, ey;
    function pointsToNormalisedVec(p,pp){
        var len;
        norm.y = pp.x - p.x;
        norm.x = -(pp.y - p.y);
        len = Math.sqrt(norm.x * norm.x + norm.y * norm.y);
        norm.x /= len;
        norm.y /= len;
        return norm;
    }
```

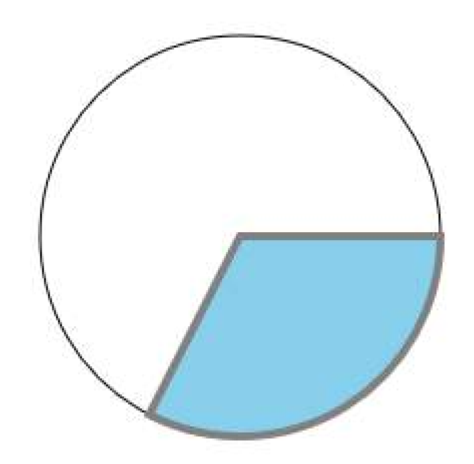
```
    var arrowWidth = arrowLength / 2;
    norm = {};
    // defaults to true for both arrows if arguments not included
    hasStartArrow = hasStartArrow === undefined || hasStartArrow === null ? true : hasStartArrow;
    hasEndArrow = hasEndArrow === undefined || hasEndArrow === null ? true : hasEndArrow;
    ctx.beginPath();
    ctx.moveTo(p0.x, p0.y);
    if (p3 === undefined) {
        ctx.quadraticCurveTo(p1.x, p1.y, p2.x, p2.y);
        ex = p2.x;  // get end point
        ey = p2.y;
        norm = pointsToNormalisedVec(p1,p2);
    } else {
        ctx.bezierCurveTo(p1.x, p1.y, p2.x, p2.y, p3.x, p3.y)
        ex = p3.x; // get end point
        ey = p3.y;
        norm = pointsToNormalisedVec(p2,p3);
    }
    if (hasEndArrow) {
        x = arrowWidth * norm.x + arrowLength * -norm.y;
        y = arrowWidth * norm.y + arrowLength * norm.x;
        ctx.moveTo(ex + x, ey + y);
        ctx.lineTo(ex, ey);
        x = arrowWidth * -norm.x + arrowLength * -norm.y;
        y = arrowWidth * -norm.y + arrowLength * norm.x;
        ctx.lineTo(ex + x, ey + y);
    }
    if (hasStartArrow) {
        norm = pointsToNormalisedVec(p0,p1);
        x = arrowWidth * norm.x - arrowLength * -norm.y;
        y = arrowWidth * norm.y - arrowLength * norm.x;
        ctx.moveTo(p0.x + x, p0.y + y);
        ctx.lineTo(p0.x, p0.y);
        x = arrowWidth * -norm.x - arrowLength * -norm.y;
        y = arrowWidth * -norm.y - arrowLength * norm.x;
        ctx.lineTo(p0.x + x, p0.y + y);
    }

    ctx.stroke();
}

function cubicCurveArrowHeads(p0, p1, p2, p3, arrowLength, hasStartArrow, hasEndArrow) {
    bezWithArrowheads(p0, p1, p2, p3, arrowLength, hasStartArrow, hasEndArrow);
}
function quadraticCurveArrowHeads(p0, p1, p2, arrowLength, hasStartArrow, hasEndArrow) {
    bezWithArrowheads(p0, p1, p2, undefined, arrowLength, hasStartArrow, hasEndArrow);
}
```

Section 15.4: Wedge

The code draws only the wedge ... circle drawn here for perspective only.

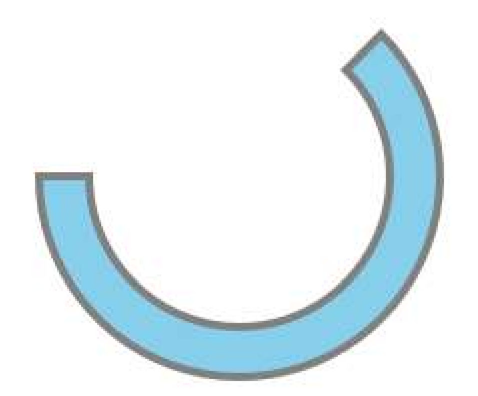


```
// Usage
var wedge={
    cx:150, cy:150,
    radius:100,
    startAngle:0,
    endAngle:Math.PI*.65
}

drawWedge(wedge,'skyblue','gray',4);

function drawWedge(w,fill,stroke,strokewidth){
    ctx.beginPath();
    ctx.moveTo(w.cx, w.cy);
    ctx.arc(w.cx, w.cy, w.radius, w.startAngle, w.endAngle);
    ctx.closePath();
    ctx.fillStyle=fill;
    ctx.fill();
    ctx.strokeStyle=stroke;
    ctx.lineWidth=strokewidth;
    ctx.stroke();
}
```

Section 15.5: Arc with both fill and stroke



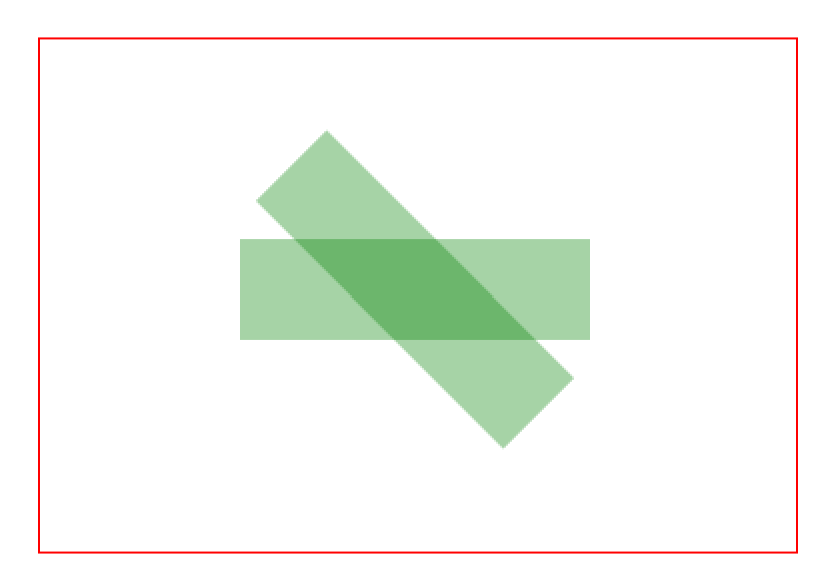
```
// Usage:
var arc={
    cx:150, cy:150,
    innerRadius:75, outerRadius:100,
    startAngle:-Math.PI/4, endAngle:Math.PI
}
```

```
drawArc(arc,'skyblue','gray',4);

function drawArc(a,fill,stroke,strokewidth){
    ctx.beginPath();
    ctx.arc(a.cx,a.cy,a.innerRadius,a.startAngle,a.endAngle);
    ctx.arc(a.cx,a.cy,a.outerRadius,a.endAngle,a.startAngle,true);
    ctx.closePath();
    ctx.fillStyle=fill;
    ctx.strokeStyle=stroke;
    ctx.lineWidth=strokewidth
    ctx.fill();
    ctx.stroke();
}
```

Chapter 16: Transformations

Section 16.1: Rotate an Image or Path around it's centerpoint



Steps#1-5 below allow any image or path-shape to be both moved anywhere on the canvas and rotated to any angle without changing any of the image/path-shape's original point coordinates.

1. Move the canvas [0,0] origin to the shape's center point

```
context.translate( shapeCenterX, shapeCenterY );
```

2. Rotate the canvas by the desired angle (in radians)

```
context.rotate( radianAngle );
```

3. Move the canvas origin back to the top-left corner

```
 context.translate( -shapeCenterX, -shapeCenterY );
```

4. Draw the image or path-shape using it's original coordinates.

```
 context.fillRect( shapeX, shapeY, shapeWidth, shapeHeight );
```

5. Always clean up! Set the transformation state back to where it was before #1

- *Step#5, Option#1:* Undo every transformation in the reverse order

```
   // undo #3
   context.translate( shapeCenterX, shapeCenterY );
   // undo #2
   context.rotate( -radianAngle );
   // undo #1
   context.translate( -shapeCenterX, shapeCenterY );
```

- *Step#5, Option#2:* If the canvas was in an untransformed state (the default) before beginning step#1, you can undo the effects of steps#1-3 by resetting all transformations to their default state

```
// set transformation to the default state (==no transformation applied)
context.setTransform(1,0,0,1,0,0)
```

Example code demo:

```
// canvas references & canvas styling
var canvas=document.createElement("canvas");
canvas.style.border='1px solid red';
document.body.appendChild(canvas);
canvas.width=378;
canvas.height=256;
var ctx=canvas.getContext("2d");
ctx.fillStyle='green';
ctx.globalAlpha=0.35;

// define a rectangle to rotate
var rect={ x:100, y:100, width:175, height:50 };

// draw the rectangle unrotated
ctx.fillRect( rect.x, rect.y, rect.width, rect.height );

// draw the rectangle rotated by 45 degrees (==PI/4 radians)
ctx.translate( rect.x+rect.width/2, rect.y+rect.height/2 );
ctx.rotate( Math.PI/4 );
ctx.translate( -rect.x-rect.width/2, -rect.y-rect.height/2 );
ctx.fillRect( rect.x, rect.y, rect.width, rect.height );
```

Section 16.2: Drawing many translated, scaled, and rotated images quickly

There are many situation where you want to draw an image that is rotated, scaled, and translated. The rotation should occur around the center of the image. This is the quickest way to do so on the 2D canvas. These functions a well suited to 2D games where the expectation is to render a few hundred even up to a 1000+ images every 60th of a second. (dependent on the hardware)

```
// assumes that the canvas context is in ctx and in scope
function drawImageRST(image, x, y, scale, rotation){
    ctx.setTransform(scale, 0, 0, scale, x, y); // set the scale and translation
    ctx.rotate(rotation);                       // add the rotation
    ctx.drawImage(image, -image.width / 2, -image.height / 2); // draw the image offset by half its
width and height
}
```

A variant can also include the alpha value which is useful for particle systems.

```
function drawImageRST_Alpha(image, x, y, scale, rotation, alpha){
    ctx.setTransform(scale, 0, 0, scale, x, y); // set the scale and translation
    ctx.rotate(rotation);                       // add the rotation
    ctx.globalAlpha = alpha;
    ctx.drawImage(image, -image.width / 2, -image.height / 2); // draw the image offset by half its
width and height
}
```

It is important to note that both functions leave the canvas context in a random state. Though the functions will not be affected other rendering my be. When you are done rendering images you may need to restore the default transform

```
ctx.setTransform(1, 0, 0, 1, 0, 0); // set the context transform back to the default
```

If you use the alpha version (second example) and then the standard version you will have to ensure that the global alpha state is restored

```
ctx.globalAlpha = 1;
```

An example of using the above functions to render some particles and the a few images

```
// assume particles to contain an array of particles
for(var i = 0; i < particles.length; i++){
    var p = particles[i];
    drawImageRST_Alpha(p.image, p.x, p.y, p.scale, p.rot, p.alpha);
    // no need to rest the alpha in the loop
}
// you need to reset the alpha as it can be any value
ctx.globalAlpha = 1;

drawImageRST(myImage, 100, 100, 1, 0.5);  // draw an image at 100,100
// no need to reset the transform
drawImageRST(myImage, 200, 200, 1, -0.5); // draw an image at 200,200
ctx.setTransform(1,0,0,1,0,0);            // reset the transform
```

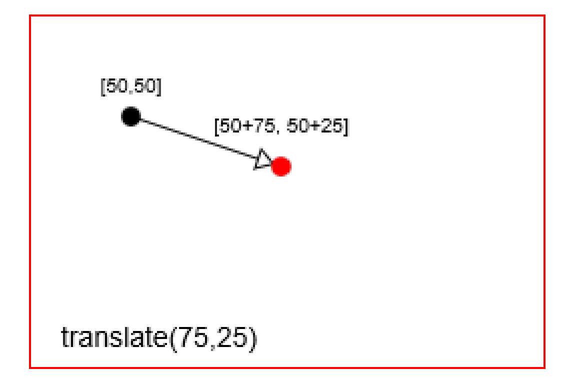
Section 16.3: Introduction to Transformations

Transformations alter a given point's starting position by moving, rotating & scaling that point.

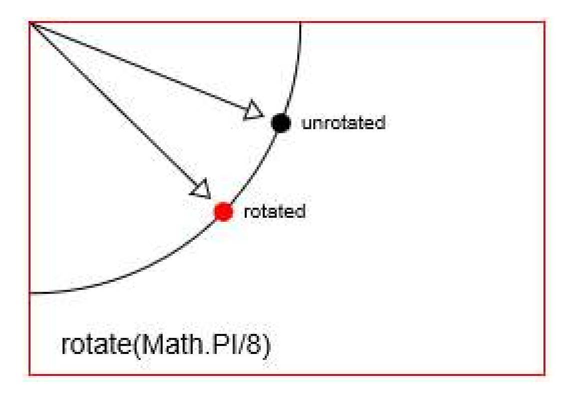
- **Translation:** Moves a point by a `distanceX` and `distanceY`.
- **Rotation:** Rotates a point by a `radian angle` around it's rotation point. The default rotation point in Html Canvas is the top-left origin [x=0,y=0] of the Canvas. But you can reposition the rotation point using translations.
- **Scaling:** Scales a point's position by a `scalingFactorX` and `scalingFactorY` from it's scaling point. The default scaling point in Html Canvas is the top-left origin [x=0,y=0] of the Canvas. But you can reposition the scaling point using translations.

You can also do less common transformations, like shearing (skewing), by directly setting the transformation matrix of the canvas using `context.transform`.

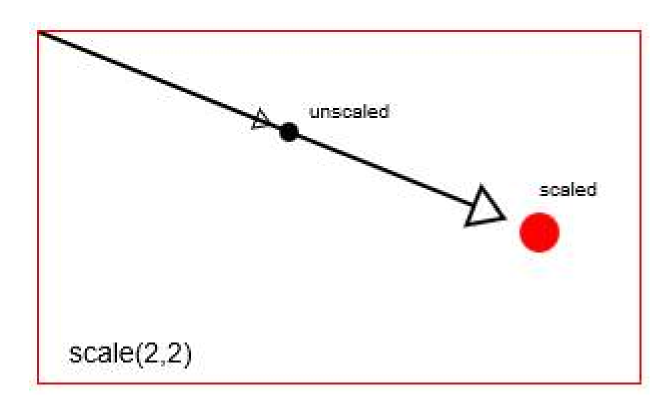
Translate (==move) a point with `context.translate(75,25)`



Rotate a point with `context.rotate(Math.PI/8)`



Scale a point with `context.scale(2,2)`



Canvas actually achieves transformations by altering the canvas' entire coordinate system.

- `context.translate` will move the canvas [0,0] origin from the top left corner to a new location.
- `context.rotate` will rotate the entire canvas coordinate system around the origin.
- `context.scale` will scale the entire canvas coordinate system around the origin. Think of this as increasing the size of every x,y on the canvas: `every x*=scaleX` and `every y*=scaleY`.

Canvas transformations are persistent. All New drawings will continue to be transformed until you reset the canvas' transformation back to it's default state (==totally untransformed). You can reset back to default with:

```
// reset context transformations to the default (untransformed) state
context.setTransform(1,0,0,1,0,0);
```

Section 16.4: A Transformation Matrix to track translated, rotated & scaled shape(s)

Canvas allows you to `context.translate`, `context.rotate` and `context.scale` in order to draw your shape in the position & size you require.

Canvas itself uses a transformation matrix to efficiently track transformations.

- You can change Canvas's matrix with `context.transform`
- You can change Canvas's matrix with individual `translate, rotate & scale` commands
- You can completely overwrite Canvas's matrix with `context.setTransform`,
- *But you can't read Canvas's internal transformation matrix -- it's write-only.*

Why use a transformation matrix?

A transformation matrix allows you to aggregate many individual translations, rotations & scalings into a single, easily reapplied matrix.

During complex animations you might apply dozens (or hundreds) of transformations to a shape. By using a transformation matrix you can (almost) instantly reapply those dozens of transformations with a single line of code.

Some Example uses:

- **Test if the mouse is inside a shape that you have translated, rotated & scaled**

  There is a built-in `context.isPointInPath` that tests if a point (eg the mouse) is inside a path-shape, but this built-in test is very slow compared to testing using a matrix.

  Efficiently testing if the mouse is inside a shape involves taking the mouse position reported by the browser and transforming it in the same way that the shape was transformed. Then you can apply hit-testing as if the shape was not transformed.

- **Redraw a shape that has been extensively translated, rotated & scaled.**

  Instead of reapplying individual transformations with multiple `.translate, .rotate, .scale` you can apply all the aggregated transformations in a single line of code.

- **Collision test shapes that have been translated, rotated & scaled**

  You can use geometry & trigonometry to calculate the points that make up transformed shapes, but it's faster to use a transformation matrix to calculate those points.

A Transformation Matrix "Class"

This code mirrors the native `context.translate`, `context.rotate`, `context.scale` transformation commands. Unlike the native canvas matrix, this matrix is readable and reusable.

Methods:

- `translate`, `rotate`, `scale` mirror the context transformation commands and allow you to feed transformations into the matrix. The matrix efficiently holds the aggregated transformations.
- `setContextTransform` takes a context and sets that context's matrix equal to this transformation matrix. This efficiently reapplies all transformations stored in this matrix to the context.
- `resetContextTransform` resets the context's transformation to it's default state (==untransformed).
- `getTransformedPoint` takes an untransformed coordinate point and converts it into a transformed point.
- `getScreenPoint` takes a transformed coordinate point and converts it into an untransformed point.
- `getMatrix` returns the aggregated transformations in the form of a matrix array.

Code:

```
var TransformationMatrix=( function(){
    // private
    var self;
    var m=[1,0,0,1,0,0];
    var reset=function(){ var m=[1,0,0,1,0,0]; }
    var multiply=function(mat){
        var m0=m[0]*mat[0]+m[2]*mat[1];
```

```
        var m1=m[1]*mat[0]+m[3]*mat[1];
        var m2=m[0]*mat[2]+m[2]*mat[3];
        var m3=m[1]*mat[2]+m[3]*mat[3];
        var m4=m[0]*mat[4]+m[2]*mat[5]+m[4];
        var m5=m[1]*mat[4]+m[3]*mat[5]+m[5];
        m=[m0,m1,m2,m3,m4,m5];
    }
    var screenPoint=function(transformedX,transformedY){
        // invert
        var d =1/(m[0]*m[3]-m[1]*m[2]);
        im=[ m[3]*d, -m[1]*d, -m[2]*d, m[0]*d, d*(m[2]*m[5]-m[3]*m[4]), d*(m[1]*m[4]-m[0]*m[5]) ];
        // point
        return({
            x:transformedX*im[0]+transformedY*im[2]+im[4],
            y:transformedX*im[1]+transformedY*im[3]+im[5]
        });
    }
    var transformedPoint=function(screenX,screenY){
        return({
            x:screenX*m[0] + screenY*m[2] + m[4],
            y:screenX*m[1] + screenY*m[3] + m[5]
        });
    }
    // public
    function TransformationMatrix(){
        self=this;
    }
    // shared methods
    TransformationMatrix.prototype.translate=function(x,y){
        var mat=[ 1, 0, 0, 1, x, y ];
        multiply(mat);
    };
    TransformationMatrix.prototype.rotate=function(rAngle){
        var c = Math.cos(rAngle);
        var s = Math.sin(rAngle);
        var mat=[ c, s, -s, c, 0, 0 ];
        multiply(mat);
    };
    TransformationMatrix.prototype.scale=function(x,y){
        var mat=[ x, 0, 0, y, 0, 0 ];
        multiply(mat);
    };
    TransformationMatrix.prototype.skew=function(radianX,radianY){
        var mat=[ 1, Math.tan(radianY), Math.tan(radianX), 1, 0, 0 ];
        multiply(mat);
    };
    TransformationMatrix.prototype.reset=function(){
        reset();
    }
    TransformationMatrix.prototype.setContextTransform=function(ctx){
        ctx.setTransform(m[0],m[1],m[2],m[3],m[4],m[5]);
    }
    TransformationMatrix.prototype.resetContextTransform=function(ctx){
        ctx.setTransform(1,0,0,1,0,0);
    }
    TransformationMatrix.prototype.getTransformedPoint=function(screenX,screenY){
        return(transformedPoint(screenX,screenY));
    }
    TransformationMatrix.prototype.getScreenPoint=function(transformedX,transformedY){
        return(screenPoint(transformedX,transformedY));
    }
    TransformationMatrix.prototype.getMatrix=function(){
```

```
        var clone=[m[0],m[1],m[2],m[3],m[4],m[5]];
        return(clone);
    }
    // return public
    return(TransformationMatrix);
})();
```

Demo:

This demo uses the Transformation Matrix "Class" above to:

- Track (==save) a rectangle's transformation matrix.
- Redraw the transformed rectangle without using context transformation commands.
- Test if the mouse has clicked inside the transformed rectangle.

Code:

```
<!doctype html>
<html>
<head>
<style>
    body{ background-color:white; }
    #canvas{border:1px solid red; }
</style>
<script>
window.onload=(function(){

    var canvas=document.getElementById("canvas");
    var ctx=canvas.getContext("2d");
    var cw=canvas.width;
    var ch=canvas.height;
    function reOffset(){
        var BB=canvas.getBoundingClientRect();
        offsetX=BB.left;
        offsetY=BB.top;
    }
    var offsetX,offsetY;
    reOffset();
    window.onscroll=function(e){ reOffset(); }
    window.onresize=function(e){ reOffset(); }

    // Transformation Matrix "Class"

    var TransformationMatrix=( function(){
        // private
        var self;
        var m=[1,0,0,1,0,0];
        var reset=function(){ var m=[1,0,0,1,0,0]; }
        var multiply=function(mat){
            var m0=m[0]*mat[0]+m[2]*mat[1];
            var m1=m[1]*mat[0]+m[3]*mat[1];
            var m2=m[0]*mat[2]+m[2]*mat[3];
            var m3=m[1]*mat[2]+m[3]*mat[3];
            var m4=m[0]*mat[4]+m[2]*mat[5]+m[4];
            var m5=m[1]*mat[4]+m[3]*mat[5]+m[5];
            m=[m0,m1,m2,m3,m4,m5];
        }
        var screenPoint=function(transformedX,transformedY){
            // invert
```

```
            var d =1/(m[0]*m[3]-m[1]*m[2]);
            im=[ m[3]*d, -m[1]*d, -m[2]*d, m[0]*d, d*(m[2]*m[5]-m[3]*m[4]), d*(m[1]*m[4]-m[0]*m[5])
];
            // point
            return({
                x:transformedX*im[0]+transformedY*im[2]+im[4],
                y:transformedX*im[1]+transformedY*im[3]+im[5]
            });
        }
        var transformedPoint=function(screenX,screenY){
            return({
                x:screenX*m[0] + screenY*m[2] + m[4],
                y:screenX*m[1] + screenY*m[3] + m[5]
            });
        }
        // public
        function TransformationMatrix(){
            self=this;
        }
        // shared methods
        TransformationMatrix.prototype.translate=function(x,y){
            var mat=[ 1, 0, 0, 1, x, y ];
            multiply(mat);
        };
        TransformationMatrix.prototype.rotate=function(rAngle){
            var c = Math.cos(rAngle);
            var s = Math.sin(rAngle);
            var mat=[ c, s, -s, c, 0, 0 ];
            multiply(mat);
        };
        TransformationMatrix.prototype.scale=function(x,y){
            var mat=[ x, 0, 0, y, 0, 0 ];
            multiply(mat);
        };
        TransformationMatrix.prototype.skew=function(radianX,radianY){
            var mat=[ 1, Math.tan(radianY), Math.tan(radianX), 1, 0, 0 ];
            multiply(mat);
        };
        TransformationMatrix.prototype.reset=function(){
            reset();
        }
        TransformationMatrix.prototype.setContextTransform=function(ctx){
            ctx.setTransform(m[0],m[1],m[2],m[3],m[4],m[5]);
        }
        TransformationMatrix.prototype.resetContextTransform=function(ctx){
            ctx.setTransform(1,0,0,1,0,0);
        }
        TransformationMatrix.prototype.getTransformedPoint=function(screenX,screenY){
            return(transformedPoint(screenX,screenY));
        }
        TransformationMatrix.prototype.getScreenPoint=function(transformedX,transformedY){
            return(screenPoint(transformedX,transformedY));
        }
        TransformationMatrix.prototype.getMatrix=function(){
            var clone=[m[0],m[1],m[2],m[3],m[4],m[5]];
            return(clone);
        }
        // return public
        return(TransformationMatrix);
    })();

    // DEMO starts here
```

```
// create a rect and add a transformation matrix
// to track it's translations, rotations & scalings
var rect={x:30,y:30,w:50,h:35,matrix:new TransformationMatrix()};

// draw the untransformed rect in black
ctx.strokeRect(rect.x, rect.y, rect.w, rect.h);
// Demo: label
ctx.font='11px arial';
ctx.fillText('Untransformed Rect',rect.x,rect.y-10);

// transform the canvas & draw the transformed rect in red
ctx.translate(100,0);
ctx.scale(2,2);
ctx.rotate(Math.PI/12);
// draw the transformed rect
ctx.strokeStyle='red';
ctx.strokeRect(rect.x, rect.y, rect.w, rect.h);
ctx.font='6px arial';
// Demo: label
ctx.fillText('Same Rect: Translated, rotated & scaled',rect.x,rect.y-6);
// reset the context to untransformed state
ctx.setTransform(1,0,0,1,0,0);

// record the transformations in the matrix
var m=rect.matrix;
m.translate(100,0);
m.scale(2,2);
m.rotate(Math.PI/12);

// use the rect's saved transformation matrix to reposition,
//     resize & redraw the rect
ctx.strokeStyle='blue';
drawTransformedRect(rect);

// Demo: instructions
ctx.font='14px arial';
ctx.fillText('Demo: click inside the blue rect',30,200);

// redraw a rect based on it's saved transformation matrix
function drawTransformedRect(r){
    // set the context transformation matrix using the rect's saved matrix
    m.setContextTransform(ctx);
    // draw the rect (no position or size changes needed!)
    ctx.strokeRect( r.x, r.y, r.w, r.h );
    // reset the context transformation to default (==untransformed);
    m.resetContextTransform(ctx);
}

// is the point in the transformed rectangle?
function isPointInTransformedRect(r,transformedX,transformedY){
    var p=r.matrix.getScreenPoint(transformedX,transformedY);
    var x=p.x;
    var y=p.y;
    return(x>r.x && x<r.x+r.w && y>r.y && y<r.y+r.h);
}

// listen for mousedown events
canvas.onmousedown=handleMouseDown;
function handleMouseDown(e){
    // tell the browser we're handling this event
    e.preventDefault();
    e.stopPropagation();
```

```
        // get mouse position
        mouseX=parseInt(e.clientX-offsetX);
        mouseY=parseInt(e.clientY-offsetY);
        // is the mouse inside the transformed rect?
        if(isPointInTransformedRect(rect,mouseX,mouseY)){
            alert('You clicked in the transformed Rect');
        }
    }

    // Demo: redraw transformed rect without using
    //       context transformation commands
    function drawTransformedRect(r,color){
        var m=r.matrix;
        var tl=m.getTransformedPoint(r.x,r.y);
        var tr=m.getTransformedPoint(r.x+r.w,r.y);
        var br=m.getTransformedPoint(r.x+r.w,r.y+r.h);
        var bl=m.getTransformedPoint(r.x,r.y+r.h);
        ctx.beginPath();
        ctx.moveTo(tl.x,tl.y);
        ctx.lineTo(tr.x,tr.y);
        ctx.lineTo(br.x,br.y);
        ctx.lineTo(bl.x,bl.y);
        ctx.closePath();
        ctx.strokeStyle=color;
        ctx.stroke();
    }

}); // end window.onload
</script>
</head>
<body>
    <canvas id="canvas" width=512 height=250></canvas>
</body>
</html>
```

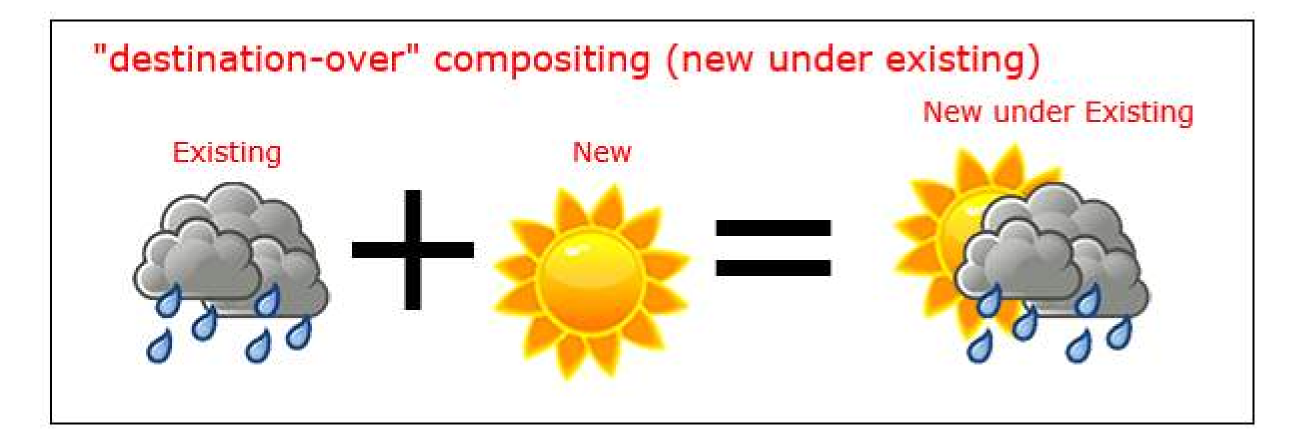
Chapter 17: Compositing

Section 17.1: Draw behind existing shapes with "destination-over"

```
context.globalCompositeOperation = "destination-over"
```

"destination-over" compositing places new drawing *under existing drawings*.

```
context.drawImage(rainy,0,0);
context.globalCompositeOperation='destination-over';  // sunny UNDER rainy
context.drawImage(sunny,0,0);
```



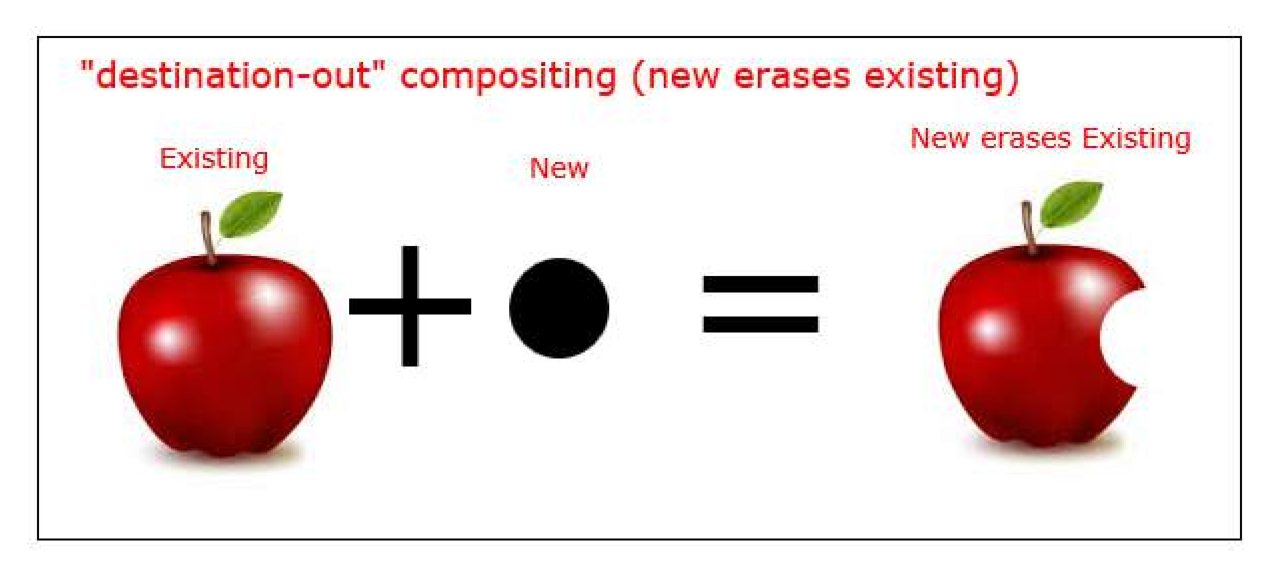
Section 17.2: Erase existing shapes with "destination-out"

```
context.globalCompositeOperation = "destination-out"
```

"destination-out" compositing uses new shapes to erase existing drawings.

The new shape is not actually drawn -- it is just used as a "cookie-cutter" to erase existing pixels.

```
context.drawImage(apple,0,0);
context.globalCompositeOperation = 'destination-out';   // bitemark erases
context.drawImage(bitemark,100,40);
```

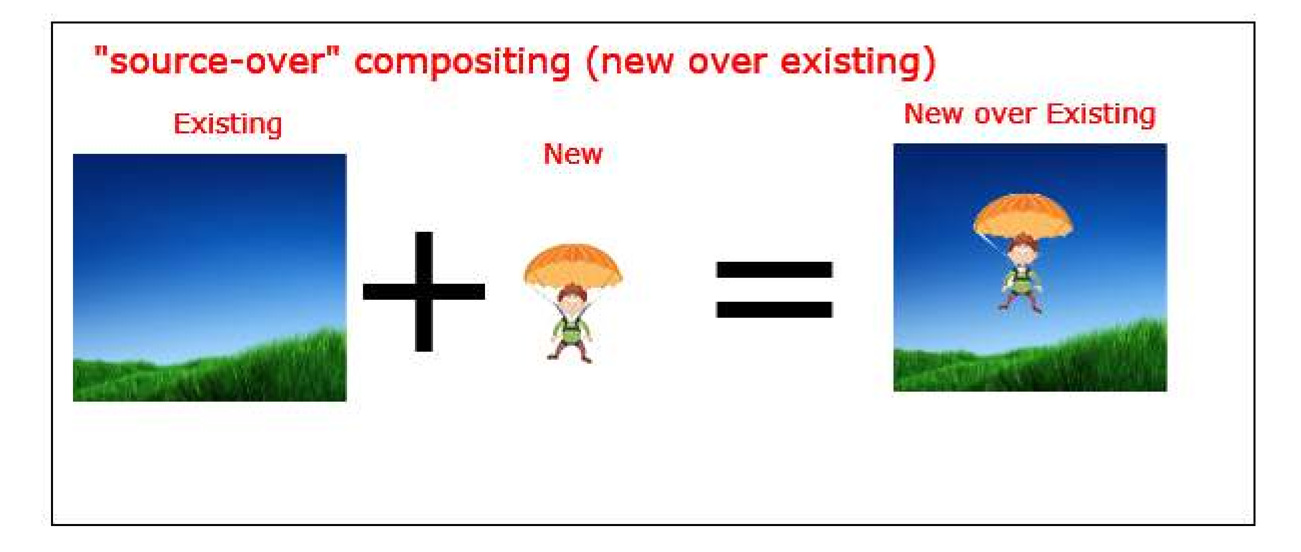


Section 17.3: Default compositing: New shapes are drawn over Existing shapes

```
context.globalCompositeOperation = "source-over"
```

"source-over" compositing **[default]**, places all new drawings over any existing drawings.

```
context.globalCompositeOperation='source-over';  // the default
context.drawImage(background,0,0);
context.drawImage(parachuter,0,0);
```



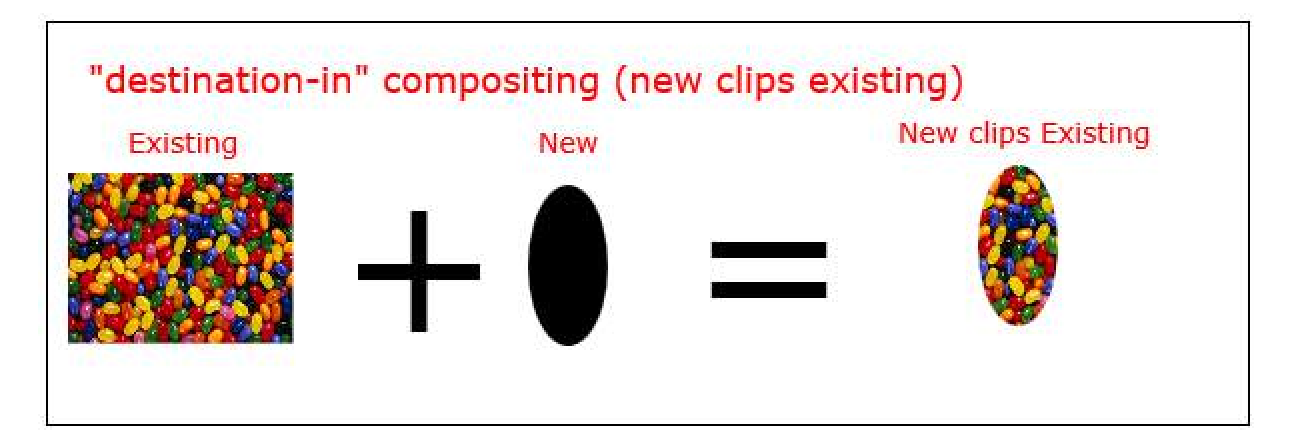
Section 17.4: Clip images inside shapes with "destination-in"

```
context.globalCompositeOperation = "destination-in"
```

"destination-in" compositing clips existing drawings inside a new shape.

Note: Any part of the existing drawing that falls outside the new drawing is erased.

```
context.drawImage(picture,0,0);
context.globalCompositeOperation='destination-in';  // picture clipped inside oval
context.drawImage(oval,0,0);
```



Section 17.5: Clip images inside shapes with "source-in"

```
context.globalCompositeOperation = "source-in";
```

`source-in` compositing clips new drawings inside an existing shape.

Note: Any part of the new drawing that falls outside the existing drawing is erased.

```
context.drawImage(oval,0,0);
context.globalCompositeOperation='source-in';  // picture clipped inside oval
context.drawImage(picture,0,0);
```

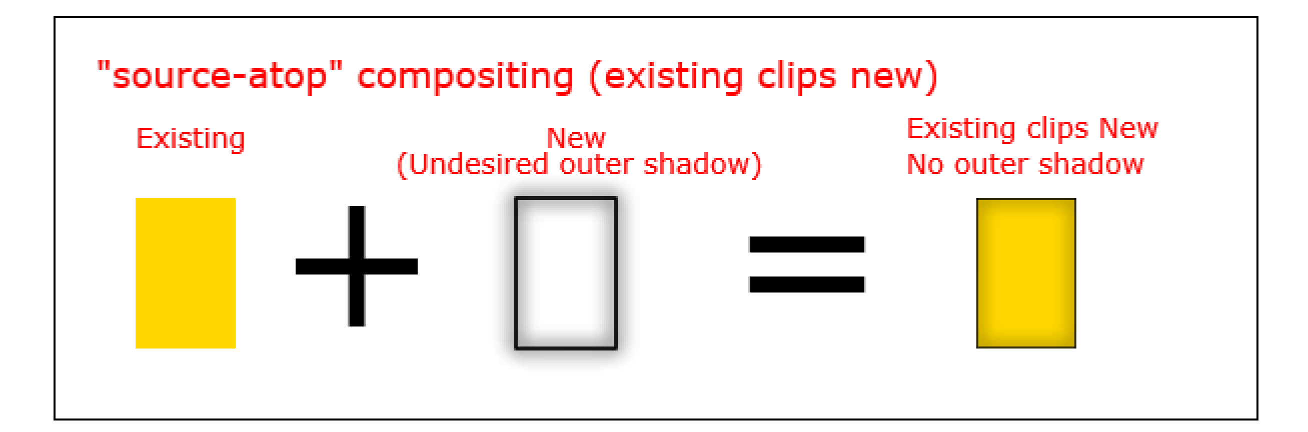


Section 17.6: Inner shadows with "source-atop"

```
context.globalCompositeOperation = 'source-atop'
```

`source-atop` compositing clips new image inside an existing shape.

```
// gold filled rect
ctx.fillStyle='gold';
ctx.fillRect(100,100,100,75);
// shadow
ctx.shadowColor='black';
ctx.shadowBlur=10;
// restrict new draw to cover existing pixels
ctx.globalCompositeOperation='source-atop';
// shadowed stroke
// "source-atop" clips off the undesired outer shadow
ctx.strokeRect(100,100,100,75);
ctx.strokeRect(100,100,100,75);
```



Section 17.7: Change opacity with "globalAlpha"

```
context.globalAlpha=0.50
```

You can change the opacity of new drawings by setting the `globalAlpha` to a value between 0.00 (fully transparent) and 1.00 (fully opaque).

The default `globalAlpha` is 1.00 (fully opaque).

Existing drawings are not affected by `globalAlpha`.

```
// draw an opaque rectangle
context.fillRect(10,10,50,50);

// change alpha to 50% -- all new drawings will have 50% opacity
context.globalAlpha=0.50;

// draw a semi-transparent rectangle
context.fillRect(100,10,50,50);
```

Section 17.8: Invert or Negate image with "difference"

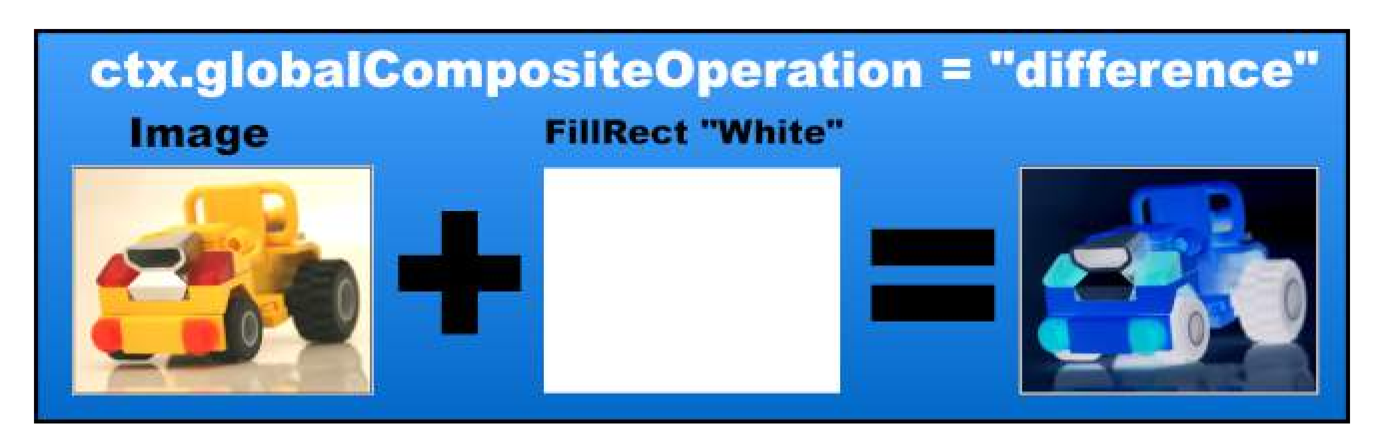
Render a white rectangle over an image with the composite operation

```
ctx.globalCompositeOperation = 'difference';
```

The amount of the effect can be controlled with the alpha setting

```
// Render the image
ctx.globalCompositeOperation='source-atop';
ctx.drawImage(image, 0, 0);

// set the composite operation
ctx.globalCompositeOperation='difference';
ctx.fillStyle = "white";
ctx.globalAlpha = alpha;  // alpha 0 = no effect 1 = full effect
ctx.fillRect(0, 0, image.width, image.height);
```



Section 17.9: Black & White with "color"

Remove color from an image via

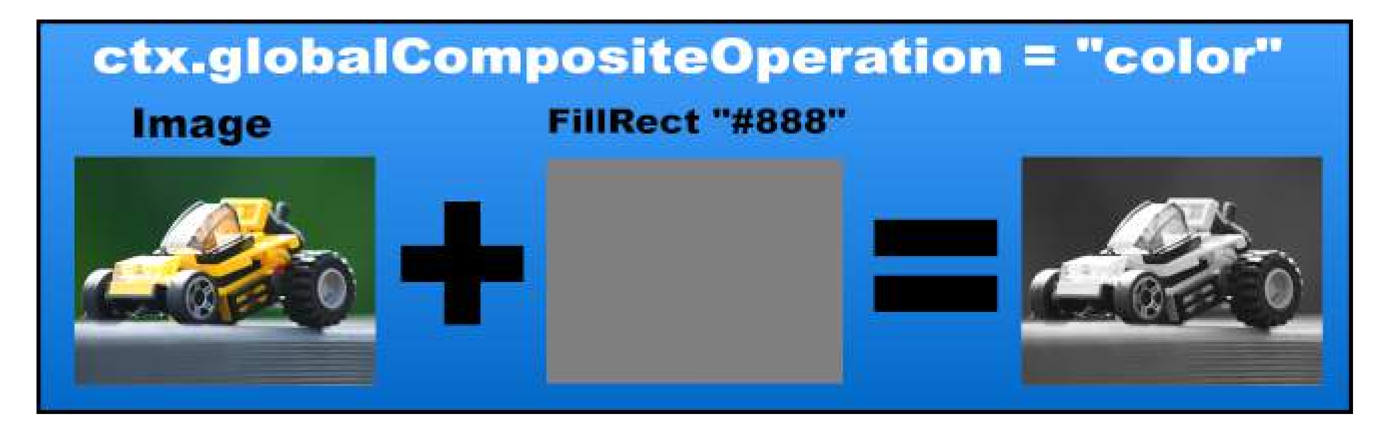
```
ctx.globalCompositeOperation = 'color';
```

The amount of the effect can be controlled with the alpha setting

```
// Render the image
ctx.globalCompositeOperation='source-atop';
```

```
ctx.drawImage(image, 0, 0);

// set the composite operation
ctx.globalCompositeOperation='color';
ctx.fillStyle = "white";
ctx.globalAlpha = alpha;  // alpha 0 = no effect 1 = full effect
ctx.fillRect(0, 0, image.width, image.height);
```



Section 17.10: Increase the color contrast with "saturation"

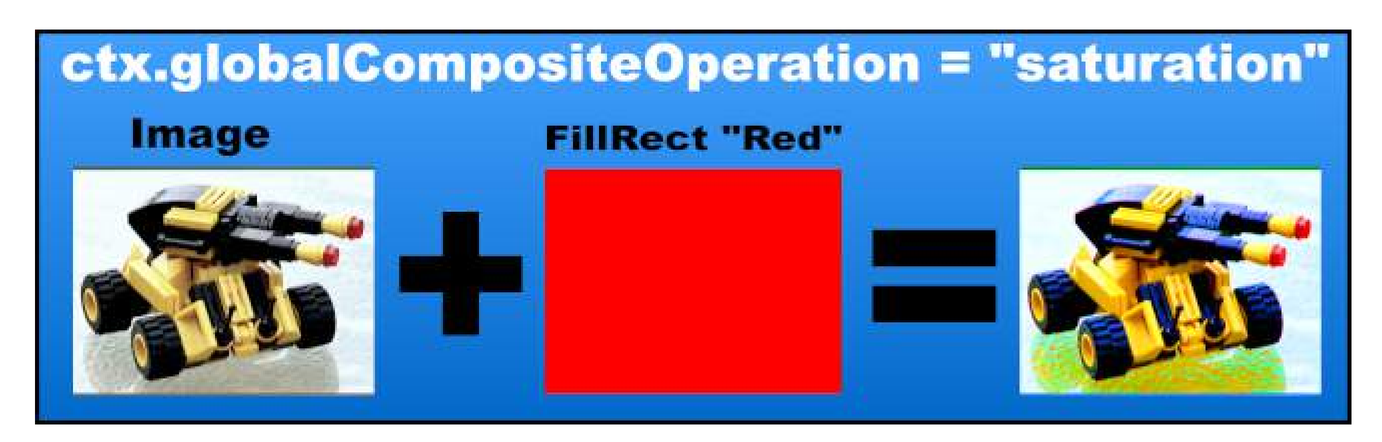
Increase the saturation level of an image with

```
ctx.globalCompositeOperation = 'saturation';
```

The amount of the effect can be controlled with the alpha setting or the amount of saturation in the fill overlay

```
// Render the image
ctx.globalCompositeOperation='source-atop';
ctx.drawImage(image, 0, 0);

// set the composite operation
ctx.globalCompositeOperation ='saturation';
ctx.fillStyle = "red";
ctx.globalAlpha = alpha;  // alpha 0 = no effect 1 = full effect
ctx.fillRect(0, 0, image.width, image.height);
```



Section 17.11: Sepia FX with "luminosity"

Create a colored sepia FX with

```
ctx.globalCompositeOperation = 'luminosity';
```

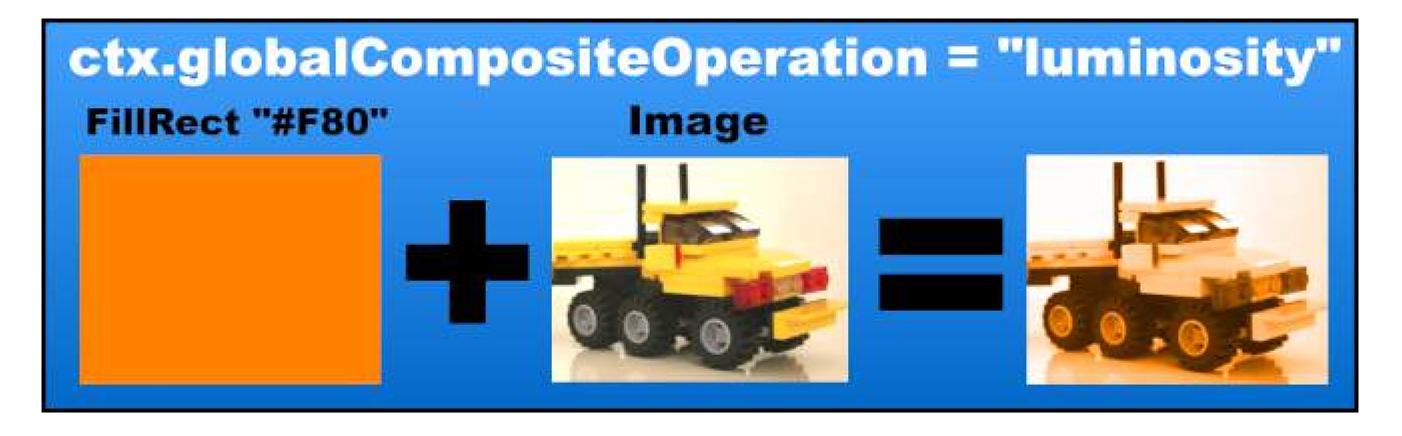
In this case the sepia colour is rendered first the the image.

The amount of the effect can be controlled with the alpha setting or the amount of saturation in the fill overlay

```
// Render the image
ctx.globalCompositeOperation='source-atop';
ctx.fillStyle = "#F80";  // the color of the sepia FX
ctx.fillRect(0, 0, image.width, image.height);

// set the composite operation
ctx.globalCompositeOperation ='luminosity';

ctx.globalAlpha = alpha;  // alpha 0 = no effect 1 = full effect
ctx.drawImage(image, 0, 0);
```



Chapter 18: Pixel Manipulation with "getImageData" and "putImageData"

Section 18.1: Introduction to "context.getImageData"

Html5 Canvas gives you the ability to fetch and change the color of any pixel on the canvas.

You can use Canvas's pixel manipulation to:

- Create a color-picker for an image or select a color on a color-wheel.
- Create complex image filters like blurring and edge detection.
- Recolor any part of an image at the pixel level (if you use HSL you can even recolor an image while retaining the important Lighting & Saturation so the result doesn't look like someone slapped paint on the image). Note: Canvas now has Blend Compositing that can also recolor an image in some cases.
- "Knockout" the background around a person/item in an image,
- Create a paint-bucket tool to detect and Floodfill part of an image (eg, change the color of a user-clicked flower petal from green to yellow).
- Examine an image for content (eg. facial recognition).

Common issues:

- For security reasons, `getImageData` is disabled if you have drawn an image originating on a different domain than the web page itself.
- `getImageData` is a relatively expensive method because it creates a large pixel-data array and because it does not use the GPU to assist its efforts. Note: Canvas now has blend compositing that can do some of the same pixel manipulation that `getImageData` does.
- For .png images, `getImageData` might not report the exact same colors as in the original .png file because the browser is allowed to do gamma-correction and alpha-premultiplication when drawing images on the canvas.

Getting pixel colors

Use `getImageData` to fetch the pixel colors for all or part of your canvas content.

The `getImageData` method returns an `imageData` object

The `imageData` object has a `.data` property that contains the pixel color information.

The `data` property is a `Uint8ClampedArray` containing the Red, Green, Blue & Alpha (opacity) color data for all requested pixels.

```
// determine which pixels to fetch (this fetches all pixels on the canvas)
var x=0;
var y=0;
var width=canvas.width;
var height=canvas.height;

// Fetch the imageData object
var imageData = context.getImageData(x,y,width,height);

// Pull the pixel color data array from the imageData object
var pixelDataArray = imageData.data;
```

You can get position of any [x,y] pixel within `data` array like this:

```
// the data[] array position for pixel [x,y]
var n = y * canvas.width + x;
```

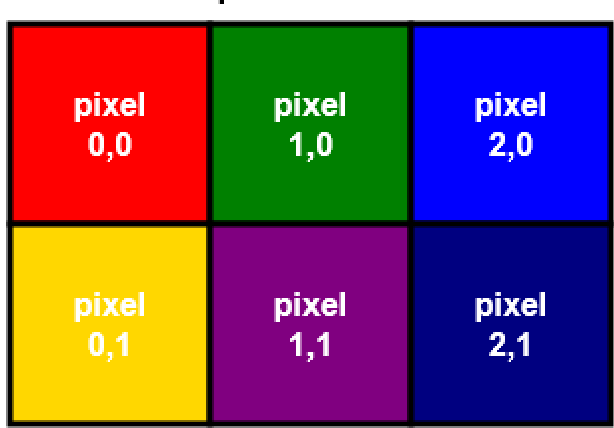
And then you can fetch that pixel's red, green, blue & alpha values like this:

```
// the RGBA info for pixel [x,y]
var red=data[n];
var green=data[n+1];
var blue=data[n+2];
var alpha=data[n+3];
```

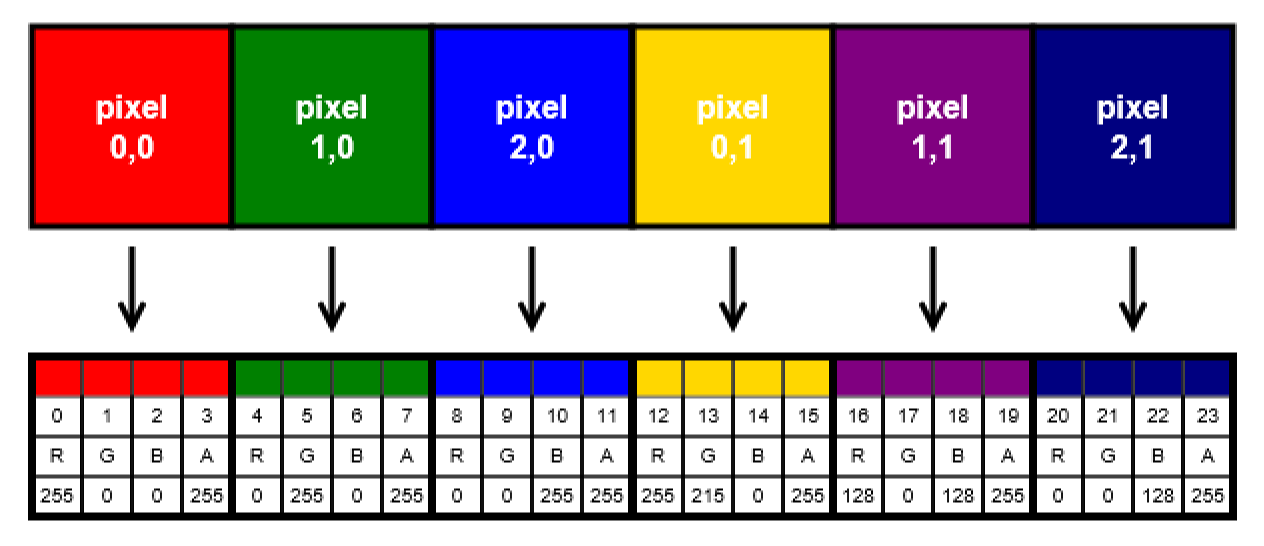
An Illustration showing how the pixel data array is structured

`context.getImageData` is illustrated below for a small 2x3 pixel sized canvas:





Pixels are arranged sequentially by row
Each pixel gets 4 array elements
(Red, Blue, Green & Alpha)



Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

almcd	Chapter 2
bjanes	Chapter 12
Blindman67	Chapters 1, 2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 15, 16 and 17
Kaiido	Chapters 1, 4, 9 and 12
markE	Chapters 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 13, 14, 15, 16, 17 and 18
Mike C	Chapter 12
Ronen Ness	Chapter 12
Spencer Wieczorek	Chapter 1
user2314737	Chapter 1

You may also like

